PDF file

# CACHE MEMORY DESIGN CONSIDERATIONS TO SUPPORT LANGUAGES WITH DYNAMIC HEAP ALLOCATION

by

Chih-Jui Peng and Gurindar S. Sohi

Computer Sciences Technical Report #860

July 1989

# CACHE MEMORY DESIGN CONSIDERATIONS TO SUPPORT LANGUAGES WITH DYNAMIC HEAP ALLOCATION

*Chih-Jui Peng and Gurindar S. Sohi*

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706

June, 1989

## Abstract

In this report, we consider the design of cache memories to support the execution of languages that make extensive use of a dynamic heap. To get insight into the cache memory design, we define several characteristics of dynamic heap references and measure these characteristics for several benchmark programs using Lisp as our model heap-intensive language. We make several observations about the heap referencing characteristics and study the implications of the referencing characteristics on cache memory design.

From our observations, we conclude that conventional cache memories are likely to be inadequate in supporting dynamic heap references. We also verify this conclusion with an extensive trace-driven simulation analysis. Then we present some cache optimizations that exploit the peculiarities of heap references. These optimizations include: i) the use of an *ALLOCATE* operation to improve the cache miss ratio as well as the data traffic ratio, ii) the use of a *biased-LRU* replacement algorithm that discriminates against garbage lines and moves the miss ratio of a cache closer to that of an unrealizable optimal cache and iii) the use of a *garbage bit* with each cache line that eliminates unnecessary write back operations.

Using trace-driven simulation, we conclude that with the heap-specific cache optimizations proposed, it is possible to design cache memories that have a miss ratio and a data traffic ratio that is close to 0. Without these optimizations, the miss ratio and data traffic ratio of a cache organization can be extremely poor, regardless of the cache size.

Two of the proposed optimizations rely on a mechanism that detects garbage soon after it is created. Since cache memory performance without the proposed optimizations is very poor, we point out the need for garbage collection mechanisms that can detect garbage almost immediately after it is created and while the garbage heap cell is still resident in the cache.

## 1. Introduction

To support the efficient execution of languages that make extensive use of a dynamic heap (such as functional languages), efficient memory support must be provided. In the execution paradigm of such languages, memory references are concentrated on 3 areas: i) an instruction or code space (which could be a part of the heap space if interpretation is used), ii) a stack space, and iii) a dynamic heap space.

Thus far, most of the work in memory system support for heap-intensive languages has concentrated on instruction and stack references (see, for example, special memory support for instruction and stack references in Lisp machines such as the Symbolics 3600 [9] ). This is justifiable since most memory references are to these 2 areas. However, as processor speeds increase, the impact of heap references on overall system performance becomes significant.

In this report, we study a topic that has largely been ignored so far in the research literature -- cache memory support for dynamic heap references. The approach we take in this study is to characterize dynamic heap references and pinpoint characteristics that impact cache memory performance. We see how certain heap reference characteristics can pose an unacceptable lower bound on the performance of conventional cache memories and verify these bound with detailed trace-driven simulation. We also see how other heap reference characteristics can be used to optimize the cache memory design. The language that we use for our studies is a dialect of Lisp.

The remainder of this report is as follows. In section 2, we define dynamic heap reference characteristics, measure these characteristics using benchmark Lisp programs, make observations about these characteristics and present the implications of the referencing characteristics on cache memory design. In section 3, we consider cache memory design using the cache miss ratio and the data traffic ratio as our performance metrics. We verify our observations of section 2 and their impact on cache memory design with extensive trace-driven simulation. We also present optimizations that are peculiar to dynamic heap references and measure the impact of these optimizations with trace-driven simulation. Many of our optimizations are based upon an efficient garbage collection strategy that is able to detect garbage while it is still resident in the cache. In section 4, we outline the requirements for such a garbage collection strategy. Finally, in section 5, we present some concluding remarks.

## 2. Characteristics of Dynamic Heap Accesses

A heap is a block of storage (*heap space*) where pieces (*objects*) are allocated in a systematic manner and deallocated in some unpredictable way. Managing the heap has been a classic problem among computer scientists since it raises an interesting question: how to create the illusion of infinite memory out of a limited amount of physical (or virtual) memory. There are two issues to be tackled: (i) how to provide this illusion correctly and (ii) how to provide this illusion efficiently. Much work has been done on both problems.

At the core of the heap management problem is *garbage collection*. Much research has been devoted to garbage collection strategies that not only manage the heap in a correct fashion, they also manage it "efficiently" [3,4,8]. Traditional metrics of efficiency have included: i) minimizing the garbage collection overhead and the interaction between garbage collection and list processing and ii) improving locality of non-garbage cells to improve paging behavior. A key component of a high-performance CPU, the cache memory, and its relationship to dynamic heap growth and management, has largely been ignored in the literature; possibly due to the lack of special purpose list processing machines with cache memories.

Our intention in this report is to investigate the relationship between dynamic heap reference patterns and cache memory design; we shall concentrate solely on this problem. The sample dynamic heap-intensive language that we use is a dialect of Lisp but we believe that our results are equally applicable to any other language that makes intensive use of a dynamic heap.

## 2.1. Definitions

Before we can pinpoint the peculiarities of heap referencing in Lisp and explain how they impact the design of cache memories to support heap accesses, some definitions are needed.

**Definition 1:** A *heap cell* is the unit of allocation in the *heap space*. It is the fundamental element for constructing *heap objects*, which are the high level abstraction of the entities seen by the program.

**Definition 2:** A *heap memory word* is a location in memory (not necessary a 4-byte word) to which a heap cell is allocated (or bound to). The collection of heap memory words form the *heap memory*.

The reason why we distinguish between *heap cells* and *heap memory words* is that a heap cell may be bound to different heap memory words at different times during the execution of a program. For example, the garbage collector may relocate a heap cell and assign it to a different heap memory word. Furthermore, the program manipulates abstract objects in the heap space whereas the processing hardware manipulates objects (contents of memory locations) in the heap memory.

**Definition 3:** The *active region of the heap space* ($R_{hs}$) is the total number of distinct heap cells referenced by a program in a trace of its execution[1].

**Definition 4:** The *active region of the heap memory* ($R_{hm}$) is the total number of distinct heap memory words referenced in a trace.

With a straightforward heap allocation algorithm and no garbage collection, $R_{hm} = 2R_{hs}$ assuming that a heap cell occupies 2 heap memory words[2]. Garbage collection can change this relation in two ways: (i) it may decrease $R_{hm}$ by reusing a collected garbage heap memory word and allocating more than one heap cell to the same heap memory word, or (ii) it may increase $R_{hm}$ by mapping the same heap cell to different heap memory words during different times in the execution of the program. Because of the difficulty in characterizing all the garbage collection algorithms in a general way and also because we are more interested in the inherent heap referencing behavior that can be exploited to improve cache performance, we shall initially assume that no garbage collection takes place. When considering cache organizations, we shall study the potential impact of garbage collection on cache performance.

**Definition 5:** The *access ratio of the heap space* ($A_{hs}$) in a trace is the average number of references made to a distinct heap cell.

**Definition 6:** The *write ratio of the heap space* ($W_{hs}$) in a trace is the average number of write accesses made to a distinct heap cell.

**Definition 7:** The *access ratio of the heap memory* ($A_{hm}$) in a trace is the average number of references made to a distinct heap memory word.

**Definition 8:** The *write ratio of the heap memory* ($W_{hm}$) in a trace is the average number of write accesses made to a distinct heap memory word.

These four definitions can be divided into two sets. The first set, $A_{hs}$ and $W_{hs}$, reflects the heap referencing behavior that is inherent in a program. In particular, the $A_{hs}$ can tell us how fast the heap space grows ($1/A_{hs}$ cell per heap reference). Because $A_{hs}$ and $W_{hs}$ are characteristics of a program, running the same program on different machine architectures will not result in different $A_{hs}$ and $W_{hs}$.

The second set, $A_{hm}$ and $W_{hm}$, reflects the memory referencing behavior observed by the processor (and memory) during the execution of a program. The resulting memory referencing behavior is a mixture of both the inherent heap space referencing behavior of a program as well

[1]Hereafter, we shall use the word *trace* to mean a trace of the execution of a program (or a part of it).

[2] Using Lisp terminology, a heap cell (*cons* cell) consists of 2 parts, a *car* and a *cdr*. If each part occupies a heap memory word, then the heap cell occupies 2 heap memory words.

as the processor implementation details such as the heap memory allocation algorithm (along with the garbage collection algorithm) and the cache configuration. In a system with no garbage collection, $A_{hs} = 2A_{hm}$ and $W_{hs} = 2W_{hm}$ because $R_{hm} = 2R_{hs}$.

$A_{hm}$ and $W_{hm}$ can provide us with insight into the performance of cache memories used to support heap memory referencing. $A_{hm}$ provides a theoretical lower bound on the cache miss ratio that can be achieved by exploiting temporal locality alone in a conventional cache, i.e., a cache without the heap-specific optimizations that we shall describe in section 3. For example, in a cache with line size equal to 1 word, because the first access to a memory word is a miss, the best-case number of hits to that word in $A_{hm}-1$, i.e., the miss ratio for access to the heap data cannot be lower than $1/A_{hm}$, *irrespective of the cache size.* By increasing the line size to $L$ words, one can exploit spatial locality and decrease the lower bound to $1/LA_{hm}$ (only 1 miss for $LA_{hm}-1$ hits).

$W_{hm}$ can also provide insight into cache performance since it allows us to compute the number of memory write requests generated and compare *write-through* and *write-back* cache organizations. More on this in sections 2.4 and 3.2.

**Definition 9:** The *temporal distance of a reference to a heap cell* ($D_c$) is the number of heap references since the last access to that cell (excluding the first access to the cell[3]). The *average temporal distance of (references to) the heap space* ($D_{hs}$) is the average of $D_c$ over all heap cells referenced.

**Definition 10:** The *temporal distance of a reference to a heap memory word* ($D_m$) is the number of heap memory references since the last access to the heap memory word (excluding the first access to the word). The *average temporal distance of (references to) the heap memory* ($D_{hm}$) is the average of $D_m$ over all heap memory words referenced.

$D_{hs}$ and $D_{hm}$ are measurements of the temporal locality in a program. Similarly to $A_{hs}$ and $W_{hs}$, $D_{hs}$ measures the locality of heap referencing inherent in a program and is a characteristic of a program while $D_{hm}$ measures the locality of heap memory referencing during the execution of a program on a particular machine and can be affected by implementation details.

**Definition 11:** The *lifetime of a heap cell* ($L_c$) in a trace is the number of heap references between the creation of the cell and the last reference made to it. The *average*

[3]The first access is excluded since it is meaningless to count the number of heap references since the last reference to a cell since there is no last reference.

*lifetime of the heap space* ($L_{hs}$) is the average of $L_c$ over all the cells referenced. $L_{hs}$ tells us how long a cell will stay "alive" (or not become garbage) on the average. It can be combined with $D_{hs}$ to estimate $A_{hs}$. Since a cell is accessed in every $D_{hs}$ references, it will be accessed $L_{hs}/D_{hs}+1$ times throughout its lifetime (the 1 is due to the first access that is not considered in $D_{hs}$), i.e., $A_{hs}=L_{hs}/D_{hs}+1$. $L_{hs}$ can also be combined with $A_{hs}$ to estimate the average number of cells that are alive simultaneously (or the average size of the *working set (WS)* of a program). Because a cell is created every $A_{hs}$ references and it stays alive for $L_{hs}$ references, there are $L_{hs}/A_{hs}$ cells alive simultaneously, on the average. Therefore, the average number of heap cells that are not garbage at a given time during the execution of a program is $WS=\dfrac{L_{hs}}{L_{hs}/D_{hs}+1}=\dfrac{D_{hs}L_{hs}}{D_{hs}+L_{hs}}$. $WS$ provides us with insight regarding the size of the cache memory that would be needed. More on this in section 2.4.

Note that we do not define *the lifetime of a heap memory word* because such definition is meaningless. A heap memory word doesn't "die," it only gets assigned to a different heap cell when the heap cell that is bound to the heap memory word becomes garbage.

## 2.2. Benchmarks and Examples

Let us illustrate our definitions with statistics collected from 8 medium size Lisp benchmark programs. Table 1 presents a description of these programs which are used in all of our remaining experiments in this report. Many of the benchmark programs that we use are obtained from Gabriel's book on Lisp benchmarks[5] and have also been used in a previous study [14].

**Table 1. Description of the Benchmark Programs Used in this Paper**

| Benchmark | Description |
|---|---|
| boyer | a theorem proving program |
| browse | a program that creates and browses through an AI-like database of unit |
| dderiv | a program for calculating derivatives of polynomials |
| destr | a program that builts and destructively traverses a tree |
| interp | a simple LISP interpreter that interprets a program for calculating the greatest common divisors using Euclid's algorithm |
| frpoly | a program on polynomial arithmetic |
| qsort | a quick-sort program |
| queen | a 8-queen problem solver |

Each of our benchmark programs is executed on a software simulator that emulates a Symbolic 3600-like architecture [9]. The simulator allows us to generate memory reference traces that are then examined by other programs. Two of the benchmarks (*boyer* and *browse*) are so long that they exceed the limit of the cache simulator (used in section 3) when the simulator uses an optimal replacement algorithm. Therefore, only a portion of these traces (100,000 heap references) are used in all our experiments. For the remaining 6 benchmarks, full traces of a complete execution of the program are used. During the generation of these traces, the garbage collector is turned off so as to satisfy our assumption of the previous section. Since no garbage collection is done, a discarded heap cell is never reused and therefore $R_{hs}$ is equal to the number of *CONS* operations in the benchmark.

Table 2 presents the total number of references, the active region, the access ratio, and the write ratio for both the heap and the stack memory areas for each of the benchmark programs. From the table, we can see that most of the memory references (about 90%) are to the stack and, therefore, stack references deserve more attention. Fortunately, we find that the active region of stack is small and the access ratio and the write ratio for stack references are very high. This suggests that any reasonable local memory support for the stack (such as a stack buffer [12] or a set of registers [14] ) should be quite effective in supporting accesses to the stack. As mentioned earlier, previous work on efficient memory support has concentrated on these stack references.

**Table 2. Characteristics of the Benchmark Programs**

| **Benchmark** | **Total References** | | **Active Region** | | **Access Ratio** | | **Write Ratio** | |
|---|---|---|---|---|---|---|---|---|
| | **Heap** | **Stack** | **Heap ($R_{hs}$)** | **Stack** | **Heap ($A_{hs}$)** | **Stack** | **Heap ($W_{hs}$)** | **Stack** |
| boyer(long) | 2056614 | 16446397 | 228664 | 173 | 8.99 | 95065.88 | 2.00 | 43928.28 |
| boyer(short) | 100000 | 801091 | 11153 | 170 | 8.97 | 4712.30 | 2.00 | 2163.94 |
| browse(long) | 2150802 | 15925997 | 221701 | 79 | 9.70 | 201594.90 | 2.18 | 92618.97 |
| browse(short) | 100000 | 741015 | 10322 | 78 | 9.69 | 9500.19 | 2.18 | 4360.27 |
| dderiv | 68200 | 492030 | 26100 | 38 | 2.61 | 12948.16 | 2.00 | 5953.02 |
| destr | 87289 | 832347 | 6889 | 45 | 12.67 | 18496.60 | 4.44 | 8252.36 |
| interp | 27002 | 224051 | 1595 | 496 | 16.93 | 451.72 | 2.00 | 202.90 |
| frpoly | 170744 | 1654670 | 14315 | 160 | 11.93 | 10341.69 | 2.16 | 4703.27 |
| qsort | 22215 | 153627 | 4695 | 611 | 4.73 | 251.44 | 2.00 | 117.04 |
| queen | 75478 | 1033397 | 4417 | 1075 | 17.09 | 961.30 | 2.00 | 425.68 |
| **Average**[4] | - | - | - | - | **8.19** | - | **2.26** | - |

[4] The two short traces are used in the calculating of the average and all subsequent cache simulations.

On the other hand, the heap references do not have this property. The active region for heap references is large and the access ratio and the write ratio are small. Once sufficient support is provided for stack accesses, adequate attention must be paid to heap accesses.

In Figure 1, we present the cumulative distribution of $D_c$, $D_m$, and $L_c$ for each of the benchmark programs. From the figure, it appears that most heap cells are short lived and most references to a heap cell occur shortly after it is created. We will come back to this later.

## 2.3. Observations

With our definitions of some key characteristics of heap referencing behavior and the data of section 2.2, we can make some observations about the behavior of heap references. Some of these observations are obvious while the others may need explanation; some of them have been exploited previously though not in the context of cache memories. Our interest is mainly in the relevance of these observations to cache memory design.

**Observation 1:** A newly allocated heap memory word is guaranteed to contain no useful data.

**Observation 2:** The first access to a new word allocated from heap (a new *cons* cell) is always a write.

**Observation 3:** Write accesses usually come in groups of two (*car* & *cdr*).

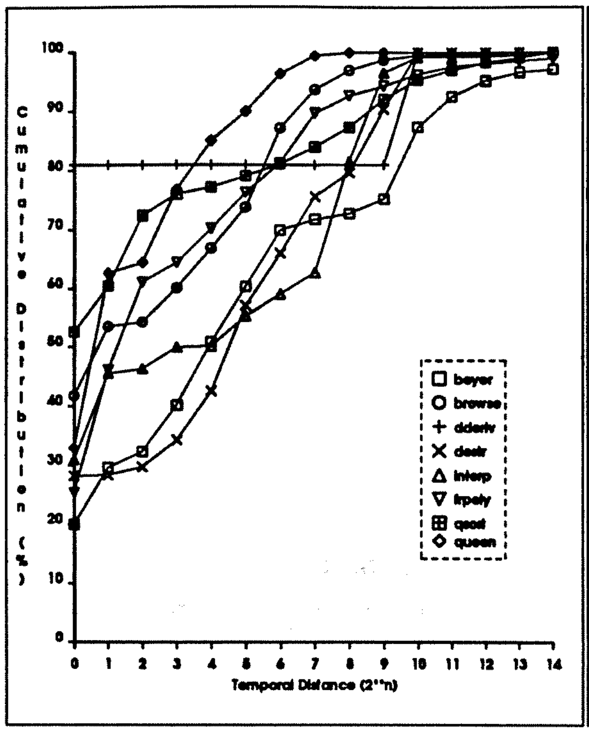
A new heap memory word contains no useful data because the heap memory is an uninitialized data area. This also explains Observation 2 that the first access to such a word must be a write to initialize it. Write accesses come in groups of two since the *car* and *cdr* components of a *cons* cell are initialized simultaneously.

**Observation 4:** The value of $A_{hs}$ is small.

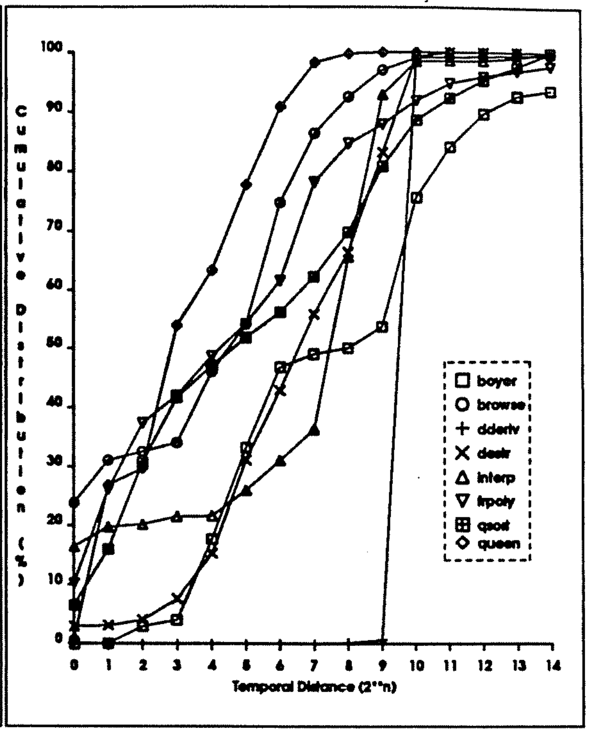
This is evident from table 2 where $A_{hs}$ for heap references ranges from 2.61 to 16.93 with an average of 8.19. The main reason for this small $A_{hs}$ is because a heap cell is associated with a fixed data structure throughout its lifetime and references to the same element in a data structure are few. On the other hand, an element in the stack can be part of different data structures at different times and consequently stack accesses can have a comparatively high access ratio (251.44 – 18496.60).

**Observation 5:** $W_{hs}$ is close to 2.

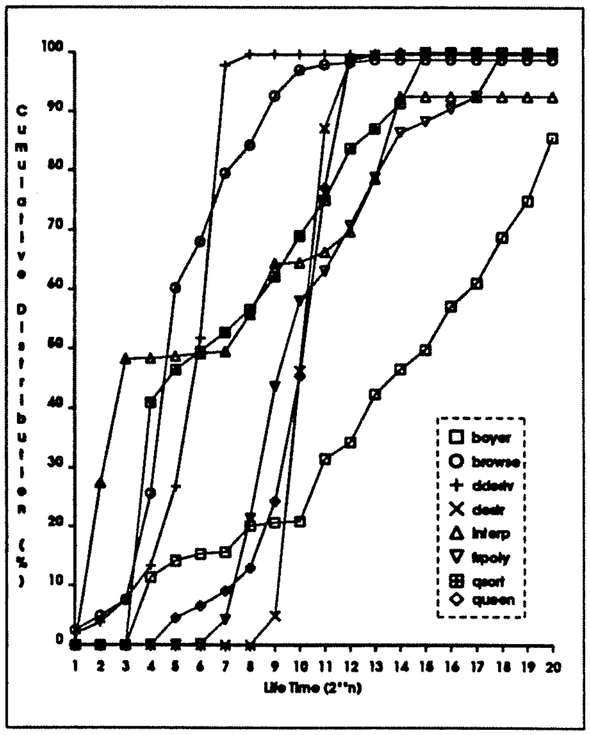
The first two write references to a *cons* cell occur when the cell is initialized. Another write reference will occur only if the *car* or the *cdr* is updated, that is, if a destructive operation such as *RPLACA* or *RPLACD* is carried out. Previous studies have shown that the frequency of



(a) Cumulative Distribution of $D_c$



(b) Cumulative Distribution of $D_m$



(c) Cumulative Distribution of $L_c$

**Figure 1. Cumulative Distributions of $D_c$, $D_m$, and $L_c$**

destructive operations is very low [2], and therefore, we can expect that $W_{hs}$ will be close to 2 (but slightly greater).

Checking with the statistics in table 2, we find that in all but one case, $W_{hs}$ is very close to 2. The only exception is *destr* which is especially written to test the speed of destructive operations and consequently has a comparatively large number of them.

**Observation 6:** A high percentage of $D_c$'s are small; or $D_{hs}$ is small.

This observation is basically the same as Clark's observation [2] where he uses *LRU-stack distance* to measure the distance between successive access to the same cell. In this report we use *temporal distance* instead because it allows us to estimate the size of the working set as shown in Definition 11.

**Observation 7:** A high percentage of $L_c$'s are very short; or $L_{hs}$ is small.

Most heap cells are allocated, referenced, and discarded (become garbage) in a short time period. Cells that do not become garbage soon generally survive for long periods of time. This observation forms the back-bone of *generation-scavenging* garbage collection algorithms [7].

## 2.4. Implications

As mentioned earlier, some of the above observations are well known and have been exploited before but not in the context of cache memories. We are mainly interested in the implications of the observations for cache memory design.

First, consider Observation 1. This observation implies that when a new heap cell is created, it should be allocated directly in the cache even if the heap memory words to which the cell is assigned do not exist in the cache. By carrying out such an *allocate* operation, we can: (i) improve the cache miss ratio and (ii) cut down on the cache-memory traffic.

Next, consider Observation 4. A small $A_{hs}$ implies a small $A_{hm}$ (if no garbage collection is done). A small $A_{hm}$ implies a high lower bound for the miss ratio of the cache. For our benchmark programs, using the formula derived from Definition 7 ($1/LA_{hm}$), the lower bound of the miss ratio for caches with line size of 1, 2, 4, and 8 words will be 24.45%, 12.22%, 6.11%, and 3.06%, respectively, *irrespective of the cache size and/or organization.* These are quite high considering cache miss ratios typically achieved for languages without intensive dynamic heap allocation [10]. In order to improve cache performance, these lower bounds must first be tackled. To reduce these lower bounds, one can not rely on locality of reference (to heap cells) alone rather one must rely on optimizations. These might include: (i) using the allocate operation described above to reduce the number of misses or (ii) artificially boosting $A_{hm}$. The value of $A_{hm}$ can be artificially boosted by assigning several heap objects to the same memory word, i.e., by identifying a heap memory word as garbage while it is still resident in the cache and assigning a new heap cell to it.

From Observation 5 ($W_{hs}$ is very close to 2), we can conclude that $W_{hm}$ is close to 1. If no garbage collection is involved, this implies that the magnitude of cache-memory traffic will be approximately the same for both *write-back* and *write-through* caches. This is an important consideration if one of the purposes of the cache memory is to reduce the cache-memory traffic and thus allow more processors to be connected together in a multiprocessor [6].

Finally, consider Observations 6 and 7. Since both $D_{hs}$ and $L_{hs}$ are small, the average working set size $WS = \frac{D_{hs}L_{hs}}{D_{hs} + L_{hs}}$ is small. For example, from Figures 1(a) and (c) we can observe that, in most cases, more than 90% of the $D_c$'s are less than 2048 heap references and a high percentage of the heap cells are discarded within 16384 heap references after allocation. Using the estimation procedure derived from Definition 11 we can roughly estimate $A_{hs}$ to be $16384/2048 + 1 = 9$, which is very close to the value we measure directly from the traces (= 8.19) in Table 1. Furthermore, the average size of the working set can be estimated to be 1820 ($\approx$ 16384*2048/(16384+2048)) cells. This means that a fully associative cache of about 3640 words, along with a good replacement algorithm that minimizes cache interference, should be adequate for capturing the locality available in the traces. Increases in the cache size beyond this is not likely to result in a substantial improvement in the miss ratio. This is in spite of the fact that the total number of heap memory words referenced by all but one of the traces is much larger than 3640 words (the smallest trace, *interp*, references 3190 heap memory words).

## 3. Cache Memory Organization

Now let us evaluate various cache organizations and see if the performance achieved is in line with the predictions made by our observations. We consider 2 practical cache organizations that represent the end points of the spectrum: (i) a direct mapped cache (DM cache) and (ii) a fully associative cache with an LRU replacement algorithm (FL cache). We are also interested in the best-case performance that can be achieved with an arbitrary cache organization. To do so, we also consider a fully associative cache with an *optimal* replacement algorithm (FO cache) such as MIN [1]. The FO cache is evaluated solely for comparison purposes; it is understood that the FO cache is impractical. In our evaluation, we shall first consider the miss ratio metric and then the data traffic ratio metric.

## 3.1. Miss Ratio

In Figure 2(a), 2(b), and 2(c), we present the average miss ratios for DM caches, FL caches, and FO caches, respectively, using our benchmark programs. Various cache sizes and line sizes are considered for each cache type. As predicted by our benchmark characteristics in section 2, all the miss ratio curves level off as the cache size becomes larger. The lower limits for the miss ratio in the FO cache for various line sizes support our observation that the miss ratio of a cache cannot be lower than $1/LA_{hm}$. The knees of the curves, again for a FO cache, support our estimate that a cache size of 3640 words is large enough to exploit the locality in the traces.

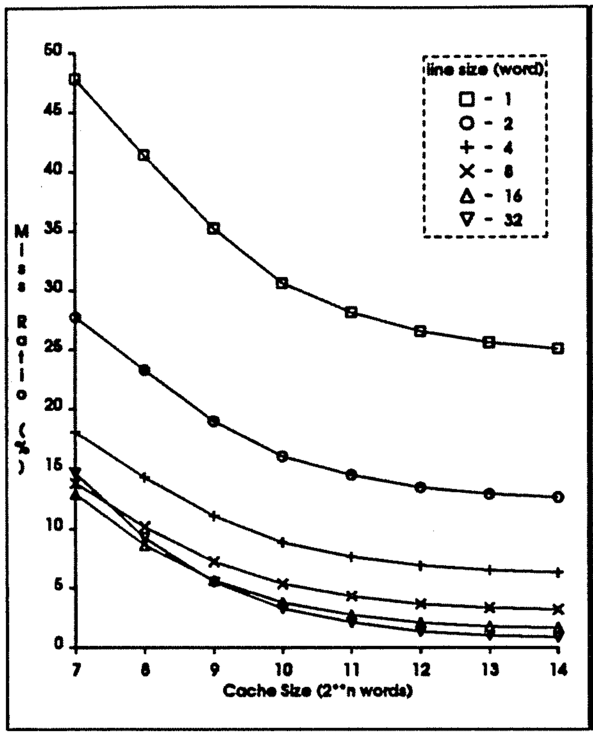
### Miss Ratio Optimization with an ALLOCATE Operation

Since increasing cache size and/or organization cannot improve the miss ratio beyond the knee points of an FO cache, we must resort to other techniques. One way to do this is to exploit Observations 1 and 2 and *allocate* the heap memory word directly in the cache without fetching it from memory on a miss. This can be accomplished with an explicit *ALLOCATE* hint (or command) to the cache. Therefore, when a *CONS* operation is carried out, the cache can be instructed to allocate the heap memory words for the *cons* cell directly in the cache without carrying out a memory read operation. By definition, the heap memory word to which the new *cons* cell is assigned is guaranteed not to be accessible by any other processor in the system before the *CONS* operation is complete, and the processor that initiates the *ALLOCATE* command can proceed just as if the *ALLOCATE* operation was a hit in the cache.

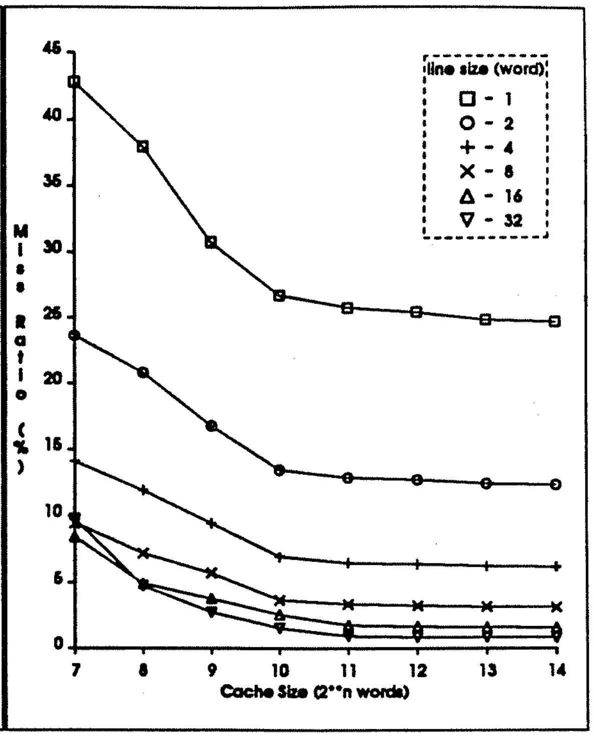
Table 3 shows the miss ratios after this *ALLOCATE* optimization. In the table, all *ALLOCATE* operations are assumed to be hits in the cache. From the table, we can see substantial improvement in the miss ratio. More importantly, the lower bound on the miss ratio has essentially been made zero for an FO cache. Further attempts at improving the miss ratio for practical cache organizations can now concentrate on other aspects, for example the replacement algorithm, in an attempt to make the performance of a practical cache approach that of the optimal cache.

### Miss Ratio Optimization with a Biased-LRU Replacement Algorithm

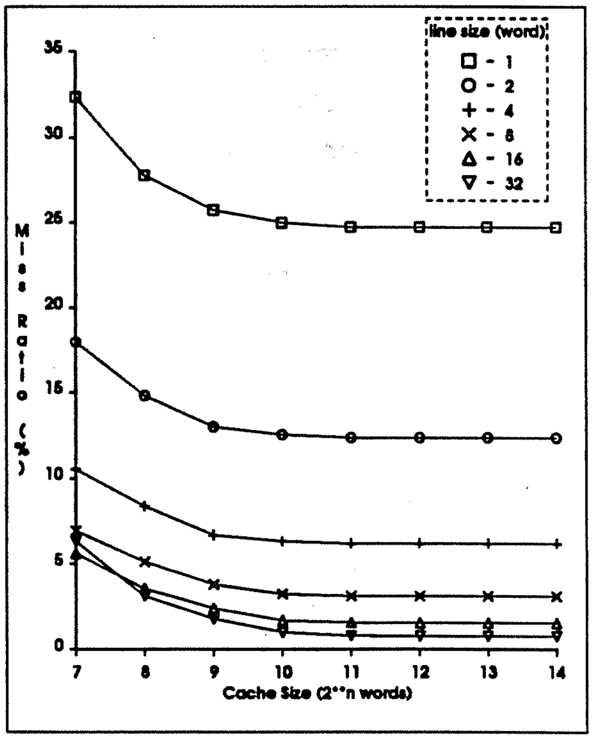
If a replacement algorithm in an associative cache always replaces a cache line that will never be referenced in the future, it will have the same performance as an unrealizable optimal replacement algorithm [13]. In most contexts, future knowledge is not possible and that is what makes implementing an optimal replacement algorithm impossible. However, for heap references, we know that a garbage heap cell by definition *will never be referenced in the future*.



(a) Miss Ratios for DM Caches



(b) Miss Ratios for FL Caches



(c) Miss Ratios for FO Caches

**Figure 2. Miss Ratios for Conventional, Non-Optimized Caches**

Therefore, if we can determine that a cache line contains only garbage, we can make that garbage line the victim of the cache replacement algorithm.

This leads us to a new cache replacement algorithm which we call the *biased-LRU* replacement algorithm. In this algorithm, an arbitrary garbage line is chosen first for replacement and if no garbage line is found, the LRU line is chosen. To detect a garbage line, we assume the presence of an additional bit in the cache state, a ***garbage bit***. We shall discuss the setting of this bit in section 4.

In Table 4, we present the miss ratios for fully-associative caches using this algorithm. Comparing this table to Table 3, we see that the miss ratios with the *biased-LRU* algorithm fall between the miss ratios of the FL and the FO caches. The improvements, however, are mainly in caches with large sizes or small lines. This is because the probability of having a garbage line is low when the cache size is small or when the line size is large.

## 3.2. Data Traffic Ratio

Now let us consider the data traffic ratio[5]. To estimate the data traffic ratios, let us carry out an analysis of the data traffic taking into account our definitions and observations of section 2. Unless mentioned otherwise, all caches are write-back caches.

The data traffic ($T$) can be divided into read data traffic ($T_r$) and write data traffic ($T_w$). In a write-back cache with a *write allocation* policy [10], $T_r$ results from fetching cache lines on misses. This can be expressed in the following equation:

$$T_r = (M_r + M_w)L \ . \tag{1}$$

where $M_r$ is the number of read misses, $M_w$ is the number of write misses, and $L$ is for the line size. The write data traffic ($T_w$), on the other hand, results from the write back of replaced lines. In a conventional cache, a replaced line needs to be written back if it is dirty. That is,

$$T_w = D\,(M_r + M_w)L \tag{2}$$

where $D$ is the probability that a replacement line is dirty. Combining equation (1) and (2), we have

[5]The data traffic ratio is defined as the ratio of the data traffic appearing on the cache-memory interconnect in the presence of a cache to the data traffic without a cache. A data traffic ratio of less than 100% implies that the traffic on the cache-memory interconnect with a cache is less than the traffic without a cache. Reducing cache-memory data traffic is crucial in the design of a shared memory multiprocessor [6].

**Table 3. Miss Ratios with an ALLOCATE Optimization**

| Line Size (words) | Cache Type | Cache Size (words) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 1 | DM | 23.44 | 16.99 | 10.87 | 6.22 | 3.75 | 2.14 | 1.18 | 0.67 |
| | FL | 18.40 | 13.55 | 6.34 | 2.29 | 1.37 | 1.06 | 0.48 | 0.30 |
| | FO | 7.94 | 3.35 | 1.31 | 0.56 | 0.29 | 0.29 | 0.29 | 0.29 |
| 2 | DM | 15.56 | 11.07 | 6.77 | 3.81 | 2.29 | 1.24 | 0.68 | 0.38 |
| | FL | 11.43 | 8.60 | 4.54 | 1.27 | 0.72 | 0.56 | 0.27 | 0.17 |
| | FO | 5.76 | 2.60 | 0.78 | 0.32 | 0.16 | 0.16 | 0.16 | 0.16 |
| 4 | DM | 11.94 | 8.19 | 4.95 | 2.74 | 1.58 | 0.83 | 0.43 | 0.23 |
| | FL | 7.99 | 5.83 | 3.36 | 0.87 | 0.41 | 0.31 | 0.16 | 0.10 |
| | FO | 4.44 | 2.27 | 0.58 | 0.23 | 0.09 | 0.09 | 0.09 | 0.09 |
| 8 | DM | 10.76 | 7.12 | 4.20 | 2.31 | 1.28 | 0.61 | 0.30 | 0.16 |
| | FL | 6.41 | 4.17 | 2.67 | 0.61 | 0.31 | 0.18 | 0.10 | 0.06 |
| | FO | 3.88 | 2.08 | 0.75 | 0.19 | 0.07 | 0.06 | 0.06 | 0.06 |
| 16 | DM | 11.36 | 7.11 | 4.15 | 2.24 | 1.20 | 0.54 | 0.24 | 0.12 |
| | FL | 6.91 | 3.41 | 2.26 | 1.01 | 0.21 | 0.10 | 0.06 | 0.03 |
| | FO | 4.10 | 2.03 | 0.88 | 0.18 | 0.05 | 0.03 | 0.03 | 0.03 |
| 32 | DM | 13.87 | 8.52 | 4.77 | 2.50 | 1.31 | 0.57 | 0.24 | 0.10 |
| | FL | 9.03 | 3.99 | 1.96 | 0.75 | 0.14 | 0.06 | 0.04 | 0.02 |
| | FO | 5.58 | 2.36 | 1.03 | 0.25 | 0.05 | 0.02 | 0.02 | 0.02 |

**Table 4. Miss Ratios with a Biased-LRU Replacement Algorithm; Fully-Associative Caches**

| Line Size (words) | Cache Size (words) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
| 1 | 17.22 | 10.18 | 3.15 | 1.81 | 0.60 | 0.40 | 0.29 | 0.29 |
| 2 | 10.75 | 6.63 | 2.21 | 1.00 | 0.33 | 0.23 | 0.16 | 0.16 |
| 4 | 7.72 | 5.00 | 2.62 | 0.66 | 0.21 | 0.15 | 0.09 | 0.09 |
| 8 | 6.29 | 3.88 | 2.10 | 0.51 | 0.20 | 0.09 | 0.06 | 0.06 |
| 16 | 6.82 | 3.23 | 2.03 | 0.95 | 0.15 | 0.06 | 0.03 | 0.03 |
| 32 | 9.02 | 3.96 | 1.88 | 0.72 | 0.12 | 0.04 | 0.02 | 0.02 |

$$T = (1 + D)(M_r + M_w)L. \qquad (3)$$

When the cache size approaches infinity ($C \rightarrow \infty$), there are no misses due to a limited capacity of the cache. This means that only the first access to a heap memory word, which is a write access, would result in a miss. Therefore, $M_r \rightarrow 0$ and $M_w \rightarrow \left\lceil \frac{R_{hm}}{L} \right\rceil$ (one miss to load $L$ words from the active region). Furthermore, since each cache line is written while it is in the cache (it must be to initialize it), $D \rightarrow 1$.

Therefore, equation (3) becomes:

$$T \rightarrow 2\left\lceil \frac{R_{hm}}{L} \right\rceil L \text{ , when } C \rightarrow \infty \tag{4}$$

Since $R_{hm}$ is much larger than $L$, $\left\lceil \frac{R_{hm}}{L} \right\rceil$ can be approximated by $\frac{R_{hm}}{L}$ and equation (4) can be further reduced to

$$T \rightarrow 2R_{hm} \text{ , when } C \rightarrow \infty. \tag{5}$$

Now, the lower bound of the data traffic ratio can be derived as:

$$\textit{Traffic Ratio} \rightarrow \frac{2\,R_{hm}}{\textit{Total References}} = \frac{2}{A_{hm}} = \frac{4}{A_{hs}}, \text{ when } C \rightarrow \infty. \tag{6}$$

From equation (6) and using $A_{hs}=8.19$ from Table 1, we can calculate the lower bound on the data traffic ratio to be 48.84%, i.e., *approximately 1 in every 2 heap references will appear as traffic on the cache-memory interconnect*. This is indeed a very surprising result considering the importance of a low data traffic ratio for shared memory multiprocessors [6], and also when comparing the observed data traffic ratio to the data traffic ratio for more traditional imperative languages [10, 11].

Does such a high value of the data traffic ratio spell doom for the execution of functional languages on multiprocessors, especially shared bus multiprocessors? We believe not since, as we shall see, we can exploit heap characteristics and optimize the data traffic ratio to the point where it is close to 0. If these suggested optimizations are not used in cache design, i.e., a conventional cache design is used, the number of processors that can be connected together to form a shared memory multiprocessor for executing heap-intensive languages will be severely restricted.

## Traffic Optimization with an ALLOCATE Operation

Let us consider the read traffic in equation (1). The read data traffic comes from either read misses (*fetch-on-read*) or write misses (*fetch-on-write*). The fetch-on-read is unavoidable because memory is the only source for the operand. However, as we saw in the case of the miss ratio, a fetch-on-write is not necessary if the heap words being fetched do not contain useful information (see Observation 1). Therefore, this component of traffic can be optimized away. In a cache with line size $L$, if a program allocates $R_{hm}$ words, $\left\lceil \frac{R_{hm}}{L} \right\rceil$ of the write misses are due to the first write to a heap word. Subtracting these misses from the $M_w$, we have

$$T_{r'} = M_r L + (M_w - \left\lceil \frac{R_{hm}}{L} \right\rceil)L. \tag{7}$$

As before, $M_r \to 0$ and $M_w \to \left\lceil \frac{R_{hm}}{L} \right\rceil$ when $C \to \infty$, therefore:

$$T_{r'} \to 0L + (\left\lceil \frac{R_{hm}}{L} \right\rceil - \left\lceil \frac{R_{hm}}{L} \right\rceil)L = 0 \tag{8}$$

and the lower bound on the total data traffic becomes

$$T = T_w \to \left\lceil \frac{R_{hm}}{L} \right\rceil L \approx R_{hm}\,. \tag{9}$$

i.e., the lower bound on the data traffic ratio is $\frac{1}{A_{hm}} = \frac{2}{A_{hs}}$.

Table 5 presents the data traffic ratios obtained from our simulation for various cache and line sizes, for both DM and FL caches. In all cases, the allocate operation is used to optimize the data traffic. Observe that as the cache size increases, the traffic ratio asymptotically approaches the lower bound $\frac{2}{A_{hs}} = 24.42\%$, irrespective of the line size. The lower bound of the data traffic is reduced by half when compared to a conventional cache without an *ALLOCATE* operation, however, the data traffic ratio is still very high when compared to the traffic ratio for imperative languages [10, 11].

## Write-Back vs. Write-Through

Before proceeding to the next optimization scheme, let us verify a prediction we made earlier regarding the data traffic ratio of write-back caches and write-through caches. In Table 6, we show the data traffic ratios of write-through caches with the *ALLOCATE* optimization. Comparing tables 5 and 6, we can see that the differences of traffic ratio between write-back and write-through caches are indeed small in all cases. If $W_{hm}$ was always 1, then write-back and write-through caches would always result in the same data traffic. *Therefore, if we are relying on write-back caches to cut down the processor/cache-memory traffic when executing languages that make intensive use of dynamic heaps, we may be in for a surprise.* Fortunately, there is a solution and let us consider that.

## Data Traffic Ratio Optimization by Recognizing Garbage

**Table 5. Data Traffic Ratios with an ALLOCATE Optimization; Write-Back Caches**

| Line Size (words) | Cache Type | Cache Size (words) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 |
| 1 | DM | 48.34 | 41.82 | 35.55 | 30.88 | 28.43 | 26.83 | 25.87 |
| | FL | 43.24 | 38.43 | 30.91 | 26.95 | 26.01 | 25.75 | 25.12 |
| 2 | DM | 58.74 | 48.83 | 38.86 | 32.59 | 29.41 | 27.27 | 26.08 |
| | FL | 50.20 | 43.75 | 33.66 | 27.19 | 26.10 | 25.84 | 25.20 |
| 4 | DM | 78.36 | 61.22 | 46.06 | 36.39 | 31.40 | 28.24 | 26.50 |
| | FL | 61.81 | 51.02 | 38.04 | 28.14 | 26.36 | 26.03 | 25.34 |
| 8 | DM | 123.95 | 90.10 | 62.25 | 45.05 | 35.93 | 30.15 | 27.32 |
| | FL | 84.70 | 63.80 | 47.54 | 29.57 | 27.28 | 26.30 | 25.57 |
| 16 | DM | 239.60 | 159.82 | 102.62 | 66.14 | 46.69 | 34.91 | 29.13 |
| | FL | 156.02 | 90.79 | 67.25 | 40.95 | 28.13 | 26.60 | 25.85 |
| 32 | DM | 557.61 | 355.86 | 212.43 | 122.34 | 75.52 | 47.36 | 33.86 |
| | FL | 378.37 | 182.88 | 105.85 | 48.75 | 29.24 | 26.96 | 26.14 |

**Table 6. Data Traffic Ratios with an ALLOCATE Optimization; Write-Through Caches**

| Line Sizr (words) | Cache Type | Cache Size (words) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 |
| 1 | DM | 50.06 | 44.08 | 38.30 | 33.76 | 31.33 | 29.75 | 28.81 |
| | FL | 45.00 | 40.47 | 33.96 | 29.92 | 29.00 | 28.69 | 28.11 |
| 2 | DM | 58.41 | 49.63 | 41.12 | 35.23 | 32.19 | 30.12 | 28.98 |
| | FL | 50.49 | 44.83 | 36.71 | 30.16 | 29.06 | 28.75 | 28.16 |
| 4 | DM | 74.88 | 60.16 | 47.34 | 38.55 | 33.92 | 30.93 | 29.34 |
| | FL | 59.57 | 50.94 | 41.07 | 31.10 | 29.26 | 28.89 | 28.26 |
| 8 | DM | 112.96 | 84.28 | 61.04 | 46.04 | 37.86 | 32.51 | 30.03 |
| | FL | 78.87 | 60.99 | 49.01 | 32.51 | 30.09 | 29.08 | 28.41 |
| 16 | DM | 208.16 | 140.88 | 93.76 | 63.38 | 46.83 | 36.29 | 31.50 |
| | FL | 137.83 | 82.16 | 63.74 | 43.85 | 30.92 | 29.28 | 28.59 |
| 32 | DM | 469.45 | 299.22 | 179.78 | 107.36 | 69.58 | 45.97 | 35.29 |
| | FL | 315.18 | 155.10 | 90.46 | 51.53 | 31.99 | 29.56 | 28.78 |

After eliminating read traffic, we now concentrate on write-back traffic. From equation (2), we see that one way to reduce the write back traffic is to reduce $D$. Unfortunately, this approach is not likely to be successful since $D \approx 1$ as argued earlier.

The other approach is to disregard the writing back of garbage cache lines. Since the garbage line contains no useful information that will be needed in the future, why bother writing them back! As before, we can identify a garbage line with a special garbage bit in the cache state. This garbage bit can be set by some mechanism that identifies garbage (see section 4) and when the cache replacement algorithm selects a line to be replaced, it does not write it back if it is garbage, even though it might be dirty.

The resulting data traffic after implementing the optimization of not writing back garbage lines is shown in Table 7. Not surprisingly, the data traffic ratio asymptotically approaches 0 as the cache size increases for all cache sizes. When compared to a conventional cache without any dynamic heap-specific optimizations, there is a 20-fold improvement in the data traffic ratio (data traffic ratios for conventional caches which we computed, but did not present simulation results for, had a lower bound of 48.84%). Given the unacceptable data traffic ratio with a conventional cache, we believe that multiprocessors designed to support the execution of dynamic heap intensive languages must make use of the suggested optimizations in the cache memories local to the processors to prevent the processor/cache-memory interconnect from severely limiting performance.

## 4. Detecting Garbage in the Cache

So far we have tacitly assumed the existence of a scheme that can efficiently detect garbage and set the garbage bit in each cache line. While the detection of garbage is not essential for correct operation, is is essential if the cache performance metrics, especially the data traffic ratio, are of concern in the processor design.

The task of garbage detection and setting the garbage bit can be carried out be any one of the several known garbage collection strategies. However, one characteristic of the selected

**Table 7. Data Traffic Ratios with all Optimizations; Write-Back Caches**

| Line Size (words) | Cache Type | Cache Size (words) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 128 | 256 | 512 | 1024 | 2048 | 4096 | 8192 |
| 1 | DM | 33.90 | 25.18 | 16.34 | 10.53 | 7.01 | 5.01 | 4.03 |
| | FL | 25.84 | 17.21 | 6.95 | 4.29 | 2.59 | 1.81 | 1.63 |
| 2 | DM | 44.43 | 32.31 | 19.75 | 12.30 | 8.03 | 5.47 | 4.24 |
| | FL | 32.40 | 21.47 | 8.29 | 4.56 | 2.66 | 1.90 | 1.68 |
| 4 | DM | 65.45 | 45.79 | 27.83 | 16.70 | 10.41 | 6.71 | 4.80 |
| | FL | 46.25 | 30.29 | 15.35 | 5.61 | 3.22 | 2.27 | 1.87 |
| 8 | DM | 112.80 | 76.00 | 45.16 | 26.08 | 15.38 | 8.91 | 5.76 |
| | FL | 71.84 | 45.69 | 23.26 | 7.73 | 4.38 | 2.65 | 2.15 |
| 16 | DM | 230.29 | 147.32 | 86.88 | 48.00 | 26.60 | 13.96 | 7.71 |
| | FL | 146.38 | 73.23 | 46.49 | 19.69 | 5.28 | 3.22 | 2.43 |
| 32 | DM | 550.59 | 345.55 | 198.91 | 105.58 | 56.04 | 26.79 | 12.60 |
| | FL | 371.90 | 170.35 | 88.61 | 28.48 | 7.12 | 3.69 | 2.66 |

garbage collection strategy that we consider especially important is the *immediacy* of garbage collection, i.e., how soon is garbage detected after it is created. This aspect of the garbage collection strategy is very important since we are relying on the strategy to collect garbage while it is still resident in the cache! In most cases, this implies garbage collection every *C CONS* operations, where *C* is the number of *cons* cells in the cache. It is also possible that the garbage collection process may interfere with the contents of the cache, possibly kicking out garbage lines from the cache and defeating the purpose of the cache optimizations. Such issues, which have largely been ignored so far in the literature on garbage collection, need to be investigated in more detail.

The garbage collection strategy that we have used in all our experiments in this report is based on reference counting. Since reference counting can detect garbage almost immediately (actually immediately if all reference counts are updated recursively when a pointer is updated), it allows us to estimate the potential benefits of our optimizations. Our garbage collection strategy assigns reference counts to heap memory words only while they are resident in the cache. This overcomes the problem of extra memory space for reference counts with each main memory word. For reasons of brevity, we do not present the details of our strategy in this report.

By detecting garbage while it is still resident in the cache, we can not only improve the data traffic ratio, we can also improve the cache miss ratio by using a better replacement algorithm and also by artificially boosting $A_{hm}$. The last point, which we have not considered in this report, allows us to improve the cache miss ratio since the same heap memory word is reused for different heap cells, all of which are cache hits. In the limit, the characteristics of heap referencing behavior can be exploited with appropriate cache optimizations (and the assist of garbage collection) to design a cache memory for a dynamically growing heap that has a miss ratio and a data traffic ratio of close to 0. Without these optimizations, if conventional caches are to be used to support dynamic heaps, they are likely to have very poor performance.

## 5. Conclusions

In this report, we have considered the design of cache memories to support the execution of languages that make intensive use of a dynamic heap. We believe most functional languages fall into this category. To facilitate our study, we defined several characteristics of heap references, measured these characteristics for Lisp (the canonical "functional" language) heap references using some benchmark programs, made some observations about the reference characteristics and studied the implications of our observations.

From our observations about heap referencing characteristics, we conclude that conventional cache memories are likely to be inadequate in supporting dynamic heap references. We verified our observations by an independent trace-driven simulation of several cache organizations. Then we presented some optimizations to enhance the performance of the cache memories. The optimizations exploit reference characteristics that are peculiar to dynamic heaps and include: i) the use of an *ALLOCATE* operation to improve the cache miss ratio as well as the data traffic ratio, ii) the use of a *biased-LRU* replacement algorithm that discriminates against garbage lines and moves the miss ratio of the cache closer to that of an unimplementable optimal cache and iii) eliminating the unnecessary write back of replaced cache lines that are garbage. With the heap-specific cache optimizations proposed, it is possible to design cache memories that have a miss ratio and a data traffic ratio that is close to 0; without the optimizations, the miss ratio and data traffic ratio of a cache organization can be extremely poor, regardless of the cache size.

Two of the proposed optimizations rely on a mechanism that detects garbage lines while the lines are still resident in the cache. This property of garbage collection algorithms, which we refer to as the *immediacy* of garbage collection, has not been studied before and needs further investigation.

## References

[1] Belady, L. A. and Gecsei, J., "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, pp. 78-101, 1966.

[2] Clark, D. W., "Measurements of Dynamic List Sturcture in Lisp," *IEEE Trans. Software Engr.*, vol. SE-5, pp. 51-59, January 1979.

[3] Cohen, J., "Garbage Collection of Linked Data Structures," *Computing Surveys*, vol. 3, pp. 341-367, September 1981.

[4] Courts, R., "Improving Locality of Reference in a Garbage-Collecting Memory Managment System," *Communications of ACM*, vol. 31, pp. 1128-1138, September 1988.

[5] Gabriel, R. P., *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

[6] Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Annual Symposium on Computer Architecture*, pp. 124-131, June 1983.

[7] Lieberman, H. and Hewitt, C., "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of ACM*, vol. 26, pp. 419-429, June 1983.

[8] Moon, D. A., "Garbage Collection in a Large Lisp System," *Proceedings 1984 ACM Symposium on Lisp and Functional Programming*, pp. 235-246, 1984.

[9] Moon, D. A., "Architecture of the Symbolic 3600," *Proceedings 12th International Symposium on Computer Architecture*, pp. 76-83, June 1985.

[10] Smith, A. J., "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473-530, Sept. 1982.

[11] Smith, A. J., "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, vol. C-36, pp. 1063-1075, September 1987.

[12] Stanley, T. J. and Wedig, R. G., "A Performance Analysis of Automatically Managed Top of Stack Buffers," in *Proceedings 14th International Symposium on Computer Architecture*, Pittsburgh, PA, pp. 272-281, June 1987.

[13] Stone, H. S., *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley, 1987.

[14] Taylor, G. S., Hilfinger, P. N., and Larus, J. R., "Evaluation of the SPUR Lisp Architecture," in *Proceedings of the 13th International Symposium on Computer Architecture*, Tokyo, Japan, pp. 444-452, June 1986.