INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame. If copyrighted materials were deleted you will find a target note listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
Thabit, Khalid Omar

CACHE MANAGEMENT BY THE COMPILER

Rice University

University Microfilms International
300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1982
RICE UNIVERSITY

CACHE MANAGEMENT BY THE COMPILER

BY

KHALID OHAR THABIT

A THESIS SUBMITTED IN THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Kenneth W. Kennedy
Prof. Kenneth W. Kennedy

Robert Thrall
Prof. Robert Thrall

J. Robert Jump
Prof. J. Robert Jump

HOUSTON, TEXAS

NOVEMBER 1981
ABSTRACT

An ideal high performance computer includes a fast processor and a multi-million byte memory of comparable speed. Since it is currently economically infeasible to have large memories with speeds matching the processor, hardware designers have included the cache. Because of its small size, and its effectiveness in eliminating the speed mismatch, the cache has become a common feature of high performance computers.

Enhancing cache performance proved to be instrumental in the speed up of cache-based computers. In most cases enhancement methods could be classified as either software based, or hardware controlled. In most cases, software based improvement methods that proved to be very effective in main memory were considered to be inapplicable to the cache. A main reason has been the cache's transparency to programs, and the fast response time of main memory. This resulted in only hardware enhancement features being considered, and implemented for the cache.

Developments in program optimization by the compiler were successful in improving the program's performance, and the understanding of program behavior. Coupling the information about a program's behavior with knowledge of the hardware structure became an good approach to optimization. With this premise we developed two cache management models: the prompting model, and the explicit management model.
Both models rely on the underlying concepts of: prefetching, clustering (packing), and loop transformations. All three are software based enhancement methods that proved to be successful in boosting main memory performance. In analyzing these methods for possible implementation in the cache we found that optimal data packing is a hard problem. Nevertheless, we suggested various heuristic methods for effective packing. We then set forth a number of conditions for loop transformations. The aim of these transformations is to facilitate prefetching (preloading) of cache blocks during loop execution.

In both models the compiler places preload requests within the program's code. These requests are serviced in parallel with program execution. Replacement decisions are determined at compile time in the explicit model, but are fully controlled by the hardware in the prompting model. In this model special tag bits are introduced to each cache block in order to facilitate replacement decisions.

The handling of aggregate data elements (arrays) are also discussed in the thesis. In the explicit model a special indexing scheme is introduced for controlling array accesses in the cache. In addition, main memory addresses are only generated for block load requests, all other addresses are for the cache.
ACKNOWLEDGEMENTS

I would like to tender thanks to my advisor, Dr. Kenneth Kennedy for suggesting my dissertation topic and for all the fruitful discussions that led to the formulation of many of the ideas proposed in the dissertation. I would further like to express my gratitude to Keith Cooper for his assistance in the preparation of this manuscript. I am also grateful to my parents and my wife for their moral support. My final acknowledgement is to King Abdul-Aziz University and the Saudi Arabian Educational Mission for their moral and financial support.
Table of Contents

I. Background

1.1 Introduction ..................................1
1.2 Cache Design ..................................6
   1.21 Direct Mapping Cache .......................7
   1.22 Associative Cache ...........................9
   1.23 Sector Cache ................................11
   1.24 Set Associative Cache ......................13
1.3 A Model for Cache Performance ...............14
   1.31 Methods for Reducing t ...............16
   1.32 Methods for Reducing P ...............17
   1.33 Methods for Reducing S ...............17
1.4 Cache Allocation Models ......................19

II. Prefetching, Clustering, and Transformation

2.1 Introduction ...................................24
2.2 Prefetching: Past and Present .............27
2.3 Clustering (Page Packing) ...................34
2.4 Program Transformation .....................41

III. Two Models For Cache Management By The Compiler

3.1 Introduction ...................................48
3.2 Data Classification and Packing ...........50
3.3 Cache Desing Requirements ..................55
3.4 The Management Models ......................57
   3.4.1 The Prompting Model .....................59
   3.4.2 The Explicit Management Model ..........64
3.5 Conclusion ....................................74

IV. Analysis of Improvement Methods

4.1 Introduction ...................................75
4.2 Packing for The One-block Cache ...........80
4.3 The Complexity of Packing ..................85
4.4 Packing for The N-Block Cache ..............95
4.5 Conditions for Loop Transformations .......100
4.6 Profitability Analysis ......................111
4.7 Conclusion ....................................123
V. Implementation Heuristics

5.1 Introduction........................................124
5.2 Optimization for a Transparent Cache .........128
  5.2.1 Loop Structured Packing for Scalars ..129
  5.2.2 Array Packing ............................133
5.3 Heuristics for the Prompting Model............140
5.3 Heuristics for the Explicit Management Model 150

VI. Conclusion and Future Directions.................155

Appendix A.............................................157

Bibliography...........................................159
CHAPTER 1

BACKGROUND

1.1. Introduction:

Since the early development of computer systems there has been a growing need for faster and more powerful computer systems. This motivated research in the areas of hardware and software development of computer systems which formed the basis of the field of study called computer science.

In early high-speed computer systems the central processing unit (CPU) and the main memory were built with circuit elements of comparable speeds. For example, the Whirlwind I, in 1953, took 8 microseconds for an add operation and 8 microseconds for a write to main memory [Kuck '78]. The Whirlwind I had a 4096 byte memory (4k) which was thought to be sufficient at the time. First, programs were written in machine language format and thus were compact. Moreover, only one program was allowed to utilize the system at any time.

The development of assemblers simplified some of the programming effort, but there was a need for a larger memory to accommodate both the assembler program and the program to be assembled. Furthermore, the introduction of the operating system improved the computer performance and simplified the programmer's task. However, part of the operating system was required to reside in main memory at all times, implying a larger memory.
Larger memories increased the cost of computer systems. To reduce this effect, designers employed slower but less expensive memory modules. These slower modules introduced a speed mismatch between main memory and the CPU.

In 1961 Kilburn, Howarth, and their associates developed the Atlas computer, the first computer with a multiprogramming operating system. Multi-programming enhanced the computer's performance by reducing the CPU idle time and increasing the system's throughput. To accommodate the large number of programs, the Atlas computer incorporated a paged memory system where each program is allocated a fraction of its address space, a number of pages, the other pages are allocated on demand [Kilburn et al. '61-'62, Howarth et al '61-'62]. As such Atlas became the first computer with virtual memory.

While hardware designers worked at improving the CPU's speed, memory designers looked for ways of providing a large amount of real memory at an economical price. Although there was a speed improvement for both the CPU and main memory, the speed gap between them has widened. One reason has been the incorporation of virtual memory which required an address calculation and a page table look-up on every memory access to check if the required page was resident in main memory. Moreover, the circuit elements used in building the CPU were faster and more expensive than those used in making the memory modules. Memory modules could be constructed using the same circuits used in building the CPU, and thus eliminate the speed gap. However, a multi-million byte memory of this type would make the computer sys-
tem prohibitively expensive. Thus the desire for a large memory conflicts with the goal of matching the speed of memory to the speed of the CPU.

To compensate for this speed gap and improve the price performance of their computer systems, designers have taken several approaches which included implementing instruction look-ahead, and increasing the number of CPU scratchpad registers. In the Univac Larc system, the processor had 99 registers [Hayes '78 cf]. The registers were used to store data elements that were frequently accessed by the program, thus decreasing the number of CPU accesses to main memory.

To speed-up the access time in virtual memory system the designers of the Atlas computer used associative memory, called look-aside registers, to store the addresses of recently accessed pages. Thus to check if a required page was in main memory the system first checks the look-aside registers. The possibility of finding an address in these registers exceeded 99% [Boon and Bunyan '76]. With such a high probability and the fast associative search, the access time of virtual memory system was reduced drastically.

In 1962, Bloom, Cohen, and Porter proposed a cost-effective design that consolidated the idea of scratchpad registers with the concept of look-aside registers [Bloom et al '62]. Their idea was to use a small, fast associative memory, called look-aside memory, as a buffer between the CPU and main memory. This idea was motivated by the observation that instructions within a loop are referenced more than once. Thus keeping the instructions for the loop in the buffer
should decrease the number of main memory references required.

In a multi-programming environment only one program would be utilizing the look-aside memory at any time. As such the size of look-aside memory need not be large, but rather have the size of an average working set of a program. With a relatively small look-aside memory, main memory seemed to be nearer to the CPU. This made the implementation of the look-aside memory cost-effective.

The look-aside memory passed through several major design phases before its actual implementation in commercial computers. Along with the design change was a name change from look-aside memory to slave memory and finally cache memory. The name cache was first introduced by a group of researchers at IBM [Conti, Gibson, and Pitkowsky '68]. Among the major computer scientist who pioneered the works on cache design were: Lee['63], Wilkes ['65], and Gibson ['67].

In his study of ways to implement the cache using associative memory, Lee noticed that data fetched from main memory may replace important data in the cache [Lee '63]. To avoid such a replacement, Lee suggested the implementation of a replacement policy algorithm which replaces stale words. Moreover, stale words that have not been modified are simply overwritten.

In 1965 Wilkes suggested an alternative design for a fast look-aside buffer, which he called slave memory [Wilkes '65]. He proposed that main memory be divided into blocks and each block of fast memory be used as slave to the memory block that is being accessed by the program. Wilkes' design eliminated the need for a replacement policy,
since every word in the memory block was to be copied to its corresponding position in the buffer block. Moreover, there was no need for an expensive associative memory and any fast memory would suffice.

One of several drawbacks of Wilkes' design was that the program and its data were required to reside within a single memory block. If a switch between blocks occurred, the content of the whole buffer would be invalidated to avoid accessing the wrong data. Excessive switching between main memory blocks, due to subroutine calls for example, would seriously degrade the system's performance.

In both the Lee and Wilkes designs, the CPU accesses the cache first; if the requested word is not found, main memory is referenced and the needed datum brought to the cache. The CPU then accesses the word from the cache.

In 1967 Gibson published a comprehensive study, conducted at IBM, of cache designs [Gibson '67]. In this study, Gibson considered the possibility of having the CPU move whole blocks of data from main memory to the cache, rather than single words as in the Lee and Wilkes designs. Gibson's work was reflected in the design of the IBM 360/85, the first IBM computer to have a cache.

Performance studies done on the cache system [Liptay '68] showed that a 16K cache would permit the computer to achieve a performance of between 66% to 94% the performance of an ideal system with all its main memory running at cache speed.
The large speed gain in memory performance due to the cache prompted many computer manufacturers to consider including cache memories in their new systems. Meanwhile, researchers concentrated on further improving the gain in memory speed from using the cache.

In the following section we shall present several of the cache designs that have been proposed. Then we shall survey various methods that enhance the performance of the memory hierarchy. This will be the topic of section three. Finally, we shall conclude the chapter with a proposal for two software models for enhancing the execution time of programs running in a cache environment.

1.2. Cache Design

The cache design has a direct effect on the cost and performance of the computer system. The various designs explored in this section have different addressing mechanisms, costs, and speeds; but they have one common feature, namely, every cache is divided into N blocks each containing n bytes, and main memory (or virtual memory) will have M of these blocks where M>>N. Moreover, the width of cache main memory bus is equal to n. In other words the block would be the smallest unit of information moved between the CPU and main memory. On the IBM 360/168, for example, the block size (n) is 64 bytes[Conners et al '79]. Hence, we shall assume that a typical cache-oriented memory system the virtual address space of the running program is made up of a number of blocks.
The address generated by the CPU is formed of two components, a main memory block number $b$ and an offset within the block $f$. For simplicity we shall assume that addresses are represented by the tuple $<b,f>$. When such an address is generated the cache is first checked for the presence of the requested block $b$, if found, then the offset $f$ is used to fetch the needed data. Otherwise the specified block $b$ must be fetched from main memory.

Implicit in all cache schemes is a method for selecting where to store a particular block and how to locate it for a memory request. Cache organizations are thus classified according to the various ways the memory blocks are assigned to the cache. Such an assignment usually adhere to a particular mapping function.

In the foregoing sections we shall survey a number of cache designs. Each design will have a different mapping function that will map the $M$ memory blocks to the $N$ cache blocks. Such a function is an inherent characteristic of the cache architecture.

For each cache block there will be a descriptor which will hold the block number of the residing memory block. When the CPU requests a datum by providing the tuple $<b,f>$, the block number $b$ is compared with the contents of the cache descriptors. If a match is found, the corresponding block will hold the required datum.

1.2.1. Direct Mapping Cache

In the direct mapping cache, the mapping function is the modulo $N$. In this case every $M/N$ memory blocks will map to one cache block.
Such blocks are said to be in the same congruence class.

For the CPU address \(<b,f>\), the cache block where this datum may reside is \(c\):

\[c = b \mod N\]

If the content of the descriptor for block \(c\) is equal to \(b\), then the offset \(f\) is used to locate the requested data. On the other hand, if no match occurs, then the block \(b\) is missing and a fault occurs. The missing block is requested from main memory, and the incoming block overwrites the content of block \(c\) (where the miss occurred). Moreover, the descriptor for block \(c\) will have a new \(b\) value. Since there is only one cache block to consider for replacement, there is no need for a replacement policy.

Direct mapping is fast, simple, and inexpensive to implement. Moreover, the cache cycle time is independent of the number of blocks in the cache. However, its main disadvantage is the frequent occurrence of contention. Contention occurs while executing a program task that requires several variables (or subroutines) that reside in two or more blocks that map to the same cache location (block). For example, doing a matrix multiplication with both matrices stored in blocks in the same congruence class would lead to contention. This would cause excessive traffic rate between main memory and cache with a corresponding loss in performance.

There is no hardware solution to avoid contention under a direct mapping cache, but there is a software remedy. The compiler could pack variables that are live and in conflict (are used at the same
time) in the same block or in blocks that end up in different congruence classes. This may be accomplished by requiring that the relative distance between conflicting blocks be less than the cache size.

Similarly, if two subroutines are called within a loop the loader can place them at a distance less than a cache size so they do not overlap each other and cause swapping.

These examples illustrate how knowledge of the cache structure and knowledge about program behavior can be coupled to achieve better cache performance, at the expense of substantial compile-time analysis.

Because of its low cost and its low hardware overhead, direct mapping cache became more attractive for use in mini computers. [Bell et al '74] did an extensive simulation studies for the PDP-8 mini-computer with a direct mapping cache.

1.2.2. Associative Cache

In contrast to the simple mapping for a direct mapping cache, an associative cache allows any of the M memory blocks to be mapped to any of the N cache blocks. Such flexibility allows for better utilization of the cache without contention.

In this design the block numbers of the current cache contents (the values in the descriptors) are placed in an associative cache directory. The value b, in the address \(<b,f>\) generated by the CPU, is searched for in the cache directory. This parallel search provides a
very fast mapping.

If the CPU finds a match, the datum is in the cache and the corresponding block is accessed. A mismatch results in a cache miss, and a fetch is issued to main memory. Since the incoming block could be placed in any cache block, one of the resident blocks is overwritten (replaced). The choice of the replacement block may cause future cache misses. With this premise, researchers have worked on various replacement policies so as to minimize this effect. Lee was the first to suggest the implementation a cache replacement policy [Lee '63]. Lee's algorithm resembled that of the least recently used (LRU).

Replacement policy algorithms have more commonly been studied in the context of paging rather than cache replacement. Nevertheless, the basic underlying concepts for these algorithms appeared to be useful for the cache environment and their implementation have become of major concern to cache designers. Among the most commonly implemented are: the first-in first-out (FIFO) method, and the least recently used (LRU) method [Smith '79].

FIFO is an aging type method where the oldest residing block will be the first to be purged. The larger the number of blocks in the cache the longer the time a block will remain in the cache before being replaced. The implementation of FIFO requires the use of a counter for each block. With each access the counters of all the blocks are incremented. When a miss occurs the block with highest value in its counter is selected for replacement. The incoming block resets its counter to zero.
In the LRU algorithm, each time a block is accessed its counter is reset to zero and the counters of the other blocks are decremented by one. At a cache miss the block having the least value is chosen for replacement. The incoming block will reset the counter to zero.

In 1966, Belady did an extensive study of various replacement policy algorithms and proved the existence of an optimal algorithm, later named Belady's MIN algorithm [Belady '66]. In the MIN algorithm the order of the pages to be referenced is required to be known and the algorithm simply selects for replacement the page furthest away from being accessed. The MIN algorithm is impossible to implement because of the a priori information it requires. It is usually used as a basis for the evaluation of newly developed replacement algorithms.

Although the associative memory has a fast access, the associative search makes it expensive to implement and the processing time taken by the replacement algorithm increases its cycle time. This makes a large associative cache unattractive.

1.2.3. Sector Cache

In this design the cache and main memory are divided into sectors, each sector made up of several blocks. A memory sector can be mapped to any cache sector. An address generated by the CPU will a triplet \( <s,b,f> \), where \( s \) is the main memory sector for the requested data. Assigned to each cache sector is an address register (descriptor) that holds the address of residing main memory sector. These
registers are grouped into an activity list with the register containing the address of the recently accessed sector at the top of the list, and the register of having the address of the least accessed sector at the bottom of the list.

When an element of a sector is requested from main memory, the sector at the bottom of the activity list is selected for replacement, and the cache sector is reassigned to the requested main memory sector. This sector is moved to the top of the activity list, and a block of that sector, which includes the requested datum, is moved to the cache. The other blocks of the main memory sector are loaded on demand (i.e. when a miss occurs).

With each block in the cache sector there is a validity bit, which is set when the block is first moved to the cache. When a miss occurs and a sector is chosen for replacement all the validity bits of its blocks are reset to zero. The requested block is then brought to the cache and its validity bit is set to one. The incoming block will occupy the same relative position as in the main memory sector. This makes the sector cache design similar to the Wilkes' slave memory design if we assume a one sector cache with each block having one word.

The sector cache performs well for programs that exhibit high locality (i.e. have its code and data within contiguous area in memory). Otherwise, only a small number of blocks will be used in each sector resulting in underutilization of the cache. For a large program underutilized cache may cause an increase in the miss ratio.
since a whole sector is invalidated just to accommodate space for one block in a different main memory sector.

The sector cache was first implemented on the IBM 360/85 [Liptay '68]. The new IBM systems, the 370's and the 3000 series, have set associative caches [Conners, et al 1979].

1.2.4. **Set Associative Cache**

A compromise design that avoids the problem of contention and the dependence of the cycle time on the cache size is the set associative cache. In this scheme $M/N$ memory blocks are mapped to a set of cache blocks rather than one block. The blocks in the same set are said to be congruent blocks (in the same congruence class).

For a given CPU address $\langle b,f \rangle$, the class address is obtained in the same way as in the direct mapping cache:

$$c = b \text{ Mod } L$$

$L$ is the number of classes and $c$ is the class that might contain the requested block. Once the required class is determined the desired block $b$ is associatively searched for among the members of the class. Since the number of blocks within a class is relatively small, the associative search is of moderate cost. On the IBM 370, for example, the set size is four [Conners, et al 1979], but only two for the Vax 780 [Vax technical summary '80].

When a cache miss occurs, replacement decisions take into account only members of the class where the miss occurred, not the whole cache. Contention will rarely occur since two blocks accessed
at the same time from the same congruence class may reside in different blocks of a set. Thus, the set associative mapping cache organization has most of the speed advantage of the direct mapping cache and much of the flexibility of the full associative cache, both at a moderate cost.

Due to its cost-performance edge, set associative cache design has been selected by many computer manufacturers when implementing a cache for their computer systems. For example, IBM, Digital Equipment and SEL all have recently chosen set associative cache designs.

In the following section we are going to investigate ways for improving cache performance in general, but since the set associative organization is so prevalent, we shall use it as our basic design model.

1.2. A Model for Cache Performance

Using a cache the average number of cycles for memory reference is reduced. The effective cycle time $T_e$ is given by [Kuck '78]:

$$T_e = t(1-S) + S \cdot T_p \quad (1.1)$$

- $t$ cache cycle time
- $T_p$ cache miss penalty
- $S$ cache miss ratio

The cache cycle time is the amount of time taken to serve a CPU request. If the requested data is not in the cache, a miss occurs. At this point a block of data (or code), that includes the requested datum, is brought from main memory to the cache. The CPU then
accesses the cache successfully.

The \textit{cache miss ratio} is the fraction of references on which a miss occurs. It is usually a function of the job mix being executed. The \textit{cache miss penalty} is the time it takes to serve a cache miss. It usually depends on the writing policy.

With a \textit{write-through policy}, main memory is updated on every write operation. On the other hand, in a \textit{write-in policy} only the the cache is modified by every write operation, and main memory is updated only when a modified block is removed from the cache.

For write-through, the CPU would write directly to the cache and to the main memory at the same time. When there are two successive writes, the second write to main memory must wait for the completion of the first write. The CPU has to stay idle for this period. This indirectly increases the cache write cycle time, which results in an increase in the effective cycle time.

On the other hand, in a write-through policy, main memory reflects all the cache updates. Thus, when a miss occurs, the requested block can overlay any block designated by the replacement policy without requiring a copy back, as in the case of the write-in policy. This means that a write-through reduces the cache miss penalty. The choice between the two writing policies depend on their contribution to the effective cycle time. In a simulation done by [Bell et al '74] the write-in policy improved the effective cycle time over the write through policy. The IBM 370/168 a write through policy while the the AMDAHL 470 V/6 has a write-in policy[ Peuto and
Shustek '77).

To simplify our analysis on cache performance we shall normalize the equation for the effective cycle time:

\[
\frac{T_e}{t} = (1-S) + \frac{(S+T_p)}{t}
\]

\[
\frac{T_e}{t} = 1 + ((T_p/t) - 1) * S
\]

If we consider that the penalty time is in terms of cache cycles, or \((T_p/t) = P\), then equation (1.1) is transformed to:

\[
T_e = t(1 + (P-1)*S) \quad (1.2)
\]

In this section shall look at various techniques that have been used to improve the effective cycle time. These methods usually aim at reducing one of the parameters involved in equation (1.2).

1.3.1. Methods for reducing \(t\)

- Using faster circuit elements [Bloch and Galage, '77].

- Employing a better cache organization.

- Having a two cache system, a data cache and an instruction cache, would reduce the cycle time since in one cache cycle the CPU could access an instruction with its operand.

- Having a pipeline control for the cache so that memory request could be initiated every machine cycle. In the IBM 370/165 the cache cycle time is equivalent to two machine cycles, but the CPU could initiates a request every machine cycle [Enslow '74].
1.3.2. Methods for reducing $P$

- Having a faster main memory

- Having an interleaved memory structure so as the loading of a cache block would require one memory cycle.

- Having a load through where the data causing the miss is directly sent to the CPU while the rest of the block is copied to the cache.

- Having a write through policy; thus at a cache miss the block selected for replacement is simply overwritten.

1.3.3. Methods for reducing $S$

- Having a larger cache [Bloch and Galage, '77].

  Choice of replacement policy [Smith '78]

- Having a write through with the condition that data to be modified which is not in the cache is not fetched; thus avoiding a cache miss [Liptay '68].

- Doing instruction look-ahead, or block prefetching in parallel with program execution.

Most of the methods implemented are hardware improvements. Nonetheless, from time to time faint software attempts have been made to organize programs to effectively use the cache [Smith '78].
Software improvement techniques are categorized as either
dynamic or static. Dynamic techniques make decisions during program
execution. The working set memory management, for main memory, is an
example of a dynamic software improvement method [Denning '68]. Here
thrashing is avoided by limiting the number of jobs in the system,
keeping track of paging activities, and making appropriate page allo-
cation decisions. These software decision are made during page
faults. Dynamic software management of the cache would not be practi-
cal due to the relatively fast response time of main memory. As such
a dynamic cache management method must be hardware implemented.
Static improvement techniques, on the other hand, are usually done
before program execution. Register allocation, and program restruc-
turing are examples of a static software improvement techniques.

Example 1.3:
Suppose a programmer writes the following Fortran loop:

```
DO 10 J=1,1024
  DO 20 I=1,8
    A(I,J) = A(I,J) + 1
  20 CONTINUE
10 CONTINUE
```
We shall assume that the cache has 4 classes with 2 blocks per class, and each block has 8 words (32 bytes). If we have an LRU policy then this loop would cause 8024 cache misses. Since the block containing $A(1,1)$ and $A(1,2)$ will be purged by the block having $A(1,8)$; thus when $J=2$, $A(1,2)$ will cause a cache miss. However if the loop is written as:

```
DO 10 I=1,8
   DO 20 J=1,1024
      A(I,J) = A(I,J) + 1
   20 CONTINUE
10 CONTINUE
```

Then only 1024 cache misses will take place. This because in Fortran arrays are stored column wise.

Such a restructuring technique could only be done by software which understands the program's behavior. There exist no hardware techniques that provide the type of improvement offered by the restructuring methods.

A further enhancement for a program running in a paged memory system is to package the program and its corresponding data into pages in such a way as to minimize interpage transitions [Kernighan '71, Baeir and Sager'76]. For example, inner loops should start at page boundaries. This way small loops would be contained within one page rather than being split between two pages.
For the cache, scalar variables should be packed into blocks in the order used rather than in the order declared. This will effectively reduce the number of blocks in the cache and consequently result in a lower miss ratio.

Prefetching and packing are the topics of the next chapter, where I present various prefetching methods implemented or suggested for paged memory systems and show how some may be applicable to the cache main memory hierarchy.

Various page packing methods are also discussed in that chapter. Most of the work on packing methods involves the code part of the program, while my study concentrates on manipulating the data part of the program for improved performance in a cache-main memory system.

1.4. Cache Allocation Models

The basic objective in this thesis is to find various compile-time techniques (software methods) for reducing the average number of cycles per memory reference or simply improve the effective cycle time:

\[ T_e = t(1 + (P-1)S) \]

Improving the cache cycle time through software methods may be difficult if not impossible; thus we shall concentrate on reducing the values of the miss ratio and the miss penalty. In particular we shall work at reducing the value of \((P-1)S\). To this end, we developed two basic methods, a preventive method and an enhancement method. For the former we look for the contributing causes of the
miss ratio then try to avoid them. In the second approach we shall develop ways for managing the cache with the aim of having a minimal number of misses.

In a set associative cache design the major contributors to the cache miss ratio are contention and the replacement of blocks that might be used again, premature replacements. We believe that either cause could be reduced by restructuring the program and by adjusting the compiler's storage allocation strategy. In particular we shall look for ways of arranging the order of data accesses such that during execution there will be less of a tendency for data elements to force one another out of the cache. Heuristics for such methods are detailed in chapter five of this thesis.

For the implementation of our second basic objective we shall capitalize on the compiler's natural capabilities for analyzing the program being compiled. The aim here is to extract vital information about the program behavior and put it in a form acceptable for the hardware which would utilize it in making a replacement or an allocation decision. Toward this objective we have considered two approaches that are reflected in two cache management models: the prompting model and the explicit management model.

In this thesis I shall discuss the theoretical and the practical aspects of building the two management models for the cache. These models will be incorporated in the compiler. Their aim will be to reduce the miss ratio and the cache miss penalty for the program being compiled.
Typical program behavior in a cache system has been recorded by Smith [79]. He shows that the number of blocks loaded is large and the number of blocks that are modified is also large. To improve cache performance would be to reduce both of these numbers or reduce their effect on cache performance.

First the compiler divides the program variables in a program into four categories then packs them into cache blocks. The four categories are:

1) Temporary variables,
2) Constants (read only variables),
3) Input variables,
4) Computed variables.

Temporary variables are those created by the compiler for expression evaluation, local identifiers, or for passing results between different sections of the program. Constants are variables whose values will not change throughout the program execution. A typical example are variables that correspond to statistical data. Input variables are those requested by an input statement or are parameters passed by a calling routine. Computed variables are those variables that are assigned a value during the program computation. Example of computed variables include variables initialized to a given constant before their use.

Since an input variable could be a constant, we shall require that we pack these variables in the order we defined them. In other words, we first pack all temporaries into cache blocks, then pack the constants, followed by the packing of the input variables and finally
pack the computed variables. Under such packing the blocks containing
the constants must be loaded to the cache, while blocks containing
computed variables need not be loaded. However, computed variables
must be copied back to main memory, while blocks holding temporary
variables need not be copied back although they have been modified.

Once the packing is complete the compiler would schedule the
loading of these blocks and the assigning of space in the cache.
These tasks would be implemented by the allocation models.

The first model is the **prompting model**. Here requests for blocks
to be loaded are passed to the hardware before they are actually
needed. When the cache-main memory bus is free then these blocks are
loaded. Program execution resumes during the prompting process. If a
miss occurs execution is halted until the miss is serviced.

The second model is called the **explicit allocation** model. For
this model the compiler takes complete control of and responsibility
for the cache management by generating instructions to allocate and
deallocate cache blocks (register blocks) within the compiled code.

The details of these models are given in the third chapter of
the thesis. Chapter two will cover previous work in prefetching,
packing, and program restructuring. The fourth chapter will cover the
theoretical aspects of cache block packing and how they affect the
allocation models. In the last chapter we shall discuss some imple-
m entation heuristics for the two models.
CHAPTER 2

Prefetching, Clustering, and Transformation

2.1. Introduction

In the previous chapter we have exposed basic concepts in cache design. Moreover, we discussed various hardware techniques which would improve the effective cycle time. The basic aim of these techniques was to reduce the cache miss ratio and the cache miss penalty. Furthermore, we discussed some software methods for improving the effective cycle time. However, to do a thorough analysis of software improvement techniques for the cache main memory hierarchy, we have to look at previous work done for the main memory secondary storage hierarchy which in many respects resembles that of the cache main memory hierarchy. The objective of these works was to improve memory utilization and reduce the CPU idle time. A major part of this chapter cover these techniques and how they came about.

In earlier computers, a user would manually fold (perform an overlay) any program larger than the available memory space. In such a scheme the program is subdivided into sections that are less than or equal to the size of the available memory. Instructions are then added to the program which instructs the hardware to either save memory contents (move_out) or overwrite the contents of memory (move_in) with the next section of the program. The execution time of the program depended heavily on its folding. Howarth, reports that 5%
of a programmer's effort is spent in coding a problem and 95% of the effort is devoted to the manual folding (overlay) [Boon and Bunyan '76]. Once a program is folded to a particular size memory, it will not run as a rule, on a smaller size memory [Sayre '69].

The introduction of automatic folding (dynamic allocation) relieved the programmer of the task of manual folding and increased his productivity. A novel method of implementing automatic folding was used in the Atlas computer, where main memory was divided into equal size sections (pages) [Howarth '62]. A program was then packed into pages.

During execution, a program would have only a number of its pages in memory at any time, if a missing page was needed, then the operating system requested that page from the secondary store. This paging scheme allowed the sharing of main memory by several programs, and increased the CPU utilization by task switching at a page fault. With the facility of scheduling and streaming of jobs, the CPU was utilized 60-80% of the time [Howarth '62]. Development in memory management and operating systems in general has been paralleled by a great enhancement in computer languages; from machine language to assembly language to high level language.

Writing a program in machine language was a difficult task that required great skill and patience. Assemblers reduced the effort of programming. When writing programs, however, the user had to understand and utilize the architectural features of the computer. Such difficulties were eliminated by the introduction of compilers which
allowed the use of high level languages that shielded the user from the details of the machine and allowed him to concentrate on solving his problem.

Although programmers' productivity was increased, programs written in assembly language took less time to execute than those written in high level language. Moreover, programs that were manually overlayed ran, in general, better than those running under dynamic allocation. This led to a debate on the importance of user productivity versus system efficiency [Boon and Bunyan '76].

To get the benefits of both worlds, language designers worked on optimizing the code generated by their compilers, while system designers looked for ways to improve their memory management systems. These techniques were aimed at reducing the page fault rate which is a major bottleneck for the system's performance. For example, an increase in page faults may lead to thrashing, where most of the CPU's time is wasted in switching between processes and no real work is accomplished.

Reducing the page fault rate for the running programs effectively diminishes the probability of thrashing, and increases the system's throughput. Like the cache miss ratio, the page fault rate depends on the amount of memory allocated to the program, the memory management (replacement policy), and the referencing pattern of the running program. Thus methods used for reducing the page fault rate may be used for avoiding cache misses, or help in the design of techniques for reducing the cache miss ratio.
Among the many enhancement techniques for reducing the page fault rate are block packing and program transformation by compilers, page clustering and prefetching by memory management systems.

2.2. Prefetching: Past and Present

In many respects prefetching and look-ahead are analogous. While look-ahead always came in the context of instructions, prefetching has been used for blocks or pages. In 1961 IBM delivered Stretch (IBM 7030), the first computer to have an instruction look-ahead facility[Suchholz '76]. Stretch was also the first pipeline computer and the look-ahead unit was essential in keeping the pipeline full.

Prefetching is the loading of pages (blocks) before the CPU requests them. Prefetching enhances systems performance if, on the average, prefetched pages are accessed before the occurrence of a fault, or the CPU does not request the access of a page displaced by a prefetched page. These are mainly influenced, by the predicting algorithm and the replacement policy, respectively.

Since the development of dynamic memory allocation, memory management has been part of the operating system. Memory management systems tend to treat all programs equally. Consequently major decisions made by the memory management are based on the statistical properties of all programs. This led early designers of prefetching schemes to investigate many possible execution behavior common to many programs.
In 1970 Joseph presented an analysis for the simulation of programs running in a paged memory [Joseph '70]. In his paper Joseph showed that for a small size memory a smaller page size results in a lower page fault rate, and a lower memory space-time product (ST). The ST is usually regarded as the sum over time of the instantaneous amount of space used by a computation. However, as the size of memory increases, the effects of smaller page size diminishes and larger pages start to show a lower page fault rate.

In the analysis of program behavior, Joseph concluded that the addressing pattern of programs are random-sequential where the address of the first instruction may be random while the address of the following instructions will likely be adjacent to the first or bear a simple relation to the first address. Moreover, operand addresses behave in a similar manner. With this premise in mind, Joseph presented two predictive algorithms which he used for a prefetching scheme.

The first algorithm was called the one block look-ahead (OBL). In the OBL at every page fault the requested page R and the page following it R+1 are fetched from secondary store. Page R is moved to main memory while the prefetched page (R+1) is loaded into a buffer and locked out; i.e. it is not accessible by the program.

If the program's next request was from page R+1, then it is removed from the buffer and made available for access, and page R+2 is requested and placed in the buffer and locked out. On the other hand, if the next request was for a page T rather than page R+1 (i.e.
a failed prediction) then page T and T+1 would be fetched and page T+1 would overwrite page R+1 in the buffer. This way prefetched pages will not displace the program's pages nor will they require more than one page of memory for the buffer.

In the second algorithm, called simple prediction (SP), failed predictions are not overwritten but left in the buffer. In this case there will be several predicted pages in the buffer requiring more memory space. However, the SP algorithm showed a better improvement in performance than the OBL, on the same programs.

Overall, the OBL and SP showed a significant improvement compared to the demand paging without prefetching. The OBL had over 25% reduction in page faults while the SP showed between 50 to 70% reduction in page faults. However, as expected, the ST factor was higher for both algorithms as compared to the ST factor of demand paging.

Baier and Sager continued on the work of Joseph by reviewing the OBL algorithm and introducing two of their own [Baier and Sager '76]. Their basic objective was to prefetch pages which share a locality and are neighbors in virtual space. First, they classified locality into two types: temporal locality and spatial locality. Then they tried to have the prefetching algorithm adapt to the changes in either type of locality.

**Temporal locality** refers to the regions of the program that are accessed during a given interval of time. While **spatial locality** refers to regions of the program that are adjacent to each other in virtual space and are accessed sequentially. Examples of temporal
locality are loops, temporary variables, working stacks or con-
stants. Arrays and straight line code are examples of spatial local-
ity.

Baier and Sager note that traditional replacement algorithms
have often been based on temporal locality. For example, in the work-
ing set policy pages that are not referenced within the last t units
of time (window size) are replaced. The LRU replaces pages that have
not been referenced for the longest period, while Belady's MIN
algorithm replaces pages that will take the longest time to their
next reference.

Two of the prefetching algorithms developed by Baier and Sager
depend on spatial locality while the third depends on temporal local-
ity. Their first algorithm is an extension of the OSL except that the
prefetched page is placed at bottom of the LRU stack and the use of a
locked out buffer is eliminated.

Their second algorithm is called spatial look ahead (SL). Here
the algorithm starts by using the prediction function:

Pred (I) = I+1.

During program execution this function would be replaced by a set of
functions depending on the access pattern of the program. If preloa-
ding was not done in the previous fault (because the page to be
preloaded was already in main memory or preloading was done in the
previous fault but the preloaded page was not referenced), then:

Pred (previous fault page) = (new page causing the fault)

This way the authors conclude that the current patterns may exist and
the predictive function should be adapted to the pattern.

The third algorithm is called temporal look ahead (TL). Here the Pred function is changed so as to reflect the temporal locality by recording the connection between the page referenced at time t-1 and the one causing the fault at time t. Baier and Sager experimented with their algorithm on three different programs: A Fortran compiler, an XPL compiler, and a one pass assembler. The probability for correct prediction varied drastically for the OBL algorithm, from 0.6 for the assembler to 0.29 for the Fortran and 0.15 for the XPL compiler.

The TL algorithm was also erratic in its behavior. However, the SL gave consistently near 0.4 correct prediction for all the programs, which shows that the SL prediction algorithm will likely give consistent results if used with other programs. Compared to the LRU, the SL had a 3 to 16% reduction in the page fault rate.

Because of its simplicity the OBL could be implemented, in hardware, as a prefetching scheme for the cache. This could be accomplished by requiring prefetching to commence just after servicing a cache miss, when the possibility of another cache miss is not likely for at least several machine cycles. On the other hand, the better performing SL algorithm is not applicable for the cache environment. First the SL prefetching algorithm entails the generation of a set of Pred functions during program execution. Since these functions vary for different programs they cannot be hardware implemented. Software implementation of the SL algorithm may result in a degradation of the
cache performance due to the fast response time of main memory.

Another type of adaptive prefetching algorithm was introduced by Franaszek and Bennett [78]. In their design secondary store is assumed to be divided into blocks of N pages. Associated with each block is a transfer number which determines the number of pages to be moved to main memory at the occurrence of a page fault to a page in that block. The requested page and those of the running program are governed by an LRU replacement policy while the prefetched pages have a FIFO replacement policy.

The number of prefetched pages that get referenced from a given block determines the new transfer number for that block. Using their adaptive prefetching on a data base and with 20% of main memory devoted for prefetched pages, Franaszek and Bennett managed to achieve significant reduction as compared to no prefetching or fixed size prefetching.

All prefetching schemes mentioned thus far have exploited the sequentiality property of programs, and have been part of the memory management system. Moreover, prefetching is done at page fault:

"For main-memory paging, page faults are currently handled by software, and without hardware support prefetching is feasible only at fault times. In addition, the overhead of prefetching for main memory includes supervisor-state CPU time to do such things as execute the page-replacement algorithm, construct a channel program to accomplish the transfer, switch tasks, and deal with the I/O interrupts when the fetch is completed.

The total overhead for transferring N (sequential) pages is only slightly higher than that for transferring one page, so sequential prefetching is a low-cost operation. Conversely, prefetching at other than faults incurs all of the costs that appear for demand paging except (usually) multiprogramming
idle. Therefore, prefetching at fault times is the only feasible strategy for software-controlled paging." [Smith '78]

In his study of prefetching, Travidi['74] suggested that prefetching requests should be initiated by the running program rather than by a prefetching scheme, and thus eliminate the possibility of prefetching an unneeded page. These ideas were the basis of manual folding, but were outmoded by the development of dynamic allocation. Nonetheless, with the advent of the optimizing compilers, manual folding concepts seem to have a comeback. In this case the Move-in instruction will be replaced by a prefetch-page instruction and the Move-out instruction could be replaced by a free-page instruction. The compiler would then place the prefetch, and the free-page instructions within the compiled program [Travidi '74, Travidi '76]. The actual preloading of the requested pages is performed at page fault.

Preloading (pages), while serving a page fault, adds little time to the page fault penalty time. Loading two consecutive pages would result in a small increase in the transfer time and will have no effect on the access time. The resulting increase would have minimal affect on the penalty time.

In the cache main memory hierarchy, the loading of two cache blocks at the same time may double the cache miss penalty. Thus for cache preloading to be beneficial, it should not be done at cache misses but during program execution by utilizing some form of cycle stealing when the cache main memory bus is free. This makes the
timing for a preload have a direct effect on the cache performance. Cache speed entails that any preloading must be hardware controlled. However, we suggest software techniques that will plan an orderly preloading scheme. Such schemes may be implemented during the optimization phase of the compiler.

2.3. Clustering (Page Packing)

Franaszek and Bennett[78] demonstrated that adaptive prefetching results in better performance than fixed policy prefetching. This means that locality varies in size during program execution. If main memory pages are made up of smaller subpages (or simply blocks) then not all these blocks tend to get accessed during any particular period of execution. Because the transfer unit between main memory and secondary store can not be less than a page, adaptive transfer loading for subpages is not possible.

One solution would be to use a smaller pages, say an average size for a subpage, and then using the adaptive prefetching scheme. Another possibility would be to cluster blocks that are likely to be accessed together into the same page. A locality is thus created which would hopefully cause a reduction in page faults.

For clustering, a program is divided into blocks of implicit statements -- statements having a unique immediate successor, the next statement in the program sequence. A flow graph is then constructed with the blocks as nodes. The graph edges are then drawn to reflect a particular relationship between the the nodes (blocks)
which would be used by the clustering algorithm.

There are basically three types of clustering methods: static clustering, dynamic off-line clustering, and dynamic on-line clustering.

An example of static clustering has been developed by Baer and Caughey[72]. In their design, information about the program structure is gathered by building the flow graph with the edges representing the flow of control between the blocks. Cycles and their embeddedness are then identified and represented by a tree with the most nested cycles as leaves and the outermost cycle as the root of the tree. The program would be represented as a forest.

Each tree is then traversed depth first by level. The traversal order of the cycles along with their sizes is passed to a clustering program which will pack the cycles into a given page size. The aim of this program is to keep the cycles within page boundaries and not to have cycles smaller than a page to be split over two pages. This is accomplished by packing cycles into a new page any time there is an overflow. Such an approach results in an increase in the number of pages for the program due to increased internal fragmentation in the pages. To reduce this affect Baer and Caughey suggested the "squeezing" of elementary cycles, which are leaves of the forest, into the end of pages and allowing them to run over pages. This may lead to the addition of extra transfer instructions within each page so as to represent the original flow of control and instruction sequence of the program.
Another static clustering technique has been suggested by Janet Fabri ['79]. Fabri's algorithm was aimed at clustering the aggregate variables in programs. Although she did not package the aggregate variables into pages, her thesis covered clustering those variables that might be accessed together, i.e., are in conflict, into the same consecutive locations, which were then overlayed by the other variables, that are not in conflict, during program execution.

Fabri's basic scheme was to represent aggregates as nodes in a graph whose edges represent the conflict relation between the variables. Two variables that are in conflict could not share the same storage area. Moreover, unlike blocks of code that appear once and are accessed sequentially in the program, aggregate variables might appear at several places throughout the program, and their accesses are usually pseudo-sequential, complicating the problem. Actually finding an optimal clustering (or overlay here) for aggregate variables is an NP-complete problem [Fabri '79 and Golombic '79].

Static clustering and manual overlay (folding) are analogous. Both require an understanding of the structure of the program in order to determine the optimal subdivision of the program for achieving the best possible performance.

As a rule of thumb, a program folded to a particular size memory will not run on a smaller size memory [Sayre '69]. On the other hand, a program packaged (clustered) to a given size page would run inefficiently on a smaller size page, because of the increase in interpage transition that has been avoided in the initial clustering. This
makes clustering in general a machine dependent optimization. If page size changes, restructuring of the program might be necessary.

In dynamic clustering statistical information about the program behavior under execution is gathered. This information is passed to the clustering algorithm which would partition the program into pages with the aim of minimizing a particular criterion that has a direct affect on the page fault rate. Examples of dynamic clustering methods have been given by Kernighan ['71], Hatfield and Gerald ['71], and Ferrari ['74].

The Kernighan algorithm is a linear time dynamic programming method that minimizes interpage transitions. However, it requires three constraints to be met.

First, the relative frequencies of transition between the blocks should be provided. Such frequencies could be gathered by executing the program once. These frequencies are then added to the edges of the graph.

The second constraint is to number the blocks (nodes) so as to reflect the instruction sequence of the program. The third constraint is that each packaged memory page must have consecutively numbered blocks. In other words, the instruction sequence within a page is the same as that given in the original program and no extra transfer instructions need to be added.

In comparing the statistical clustering of Baer and Gaughey to that of Kernighan, Baer found that the former gives lower value for
interpage transition but without holding to Kernighan's third condition [Baer '78].

In studying program behavior in virtual memory systems, Hatfield and Gerald found a clustering algorithm for improving program performance. Basically, they divided the program into blocks and looked at the reference string of the blocks as generated by the running program. Using the reference string they built the nearness matrix whose entries reflect the number of times blocks neighbor each other in the reference string.

For a given page size the clustering algorithm would pack the program blocks so as to maximize the sum of the nearness when taken over the pages. In other words, three blocks in the same pages would result in the sum of their entry pairs as recorded in the nearness matrix. The clustering algorithm simply did row and column manipulation on the nearness matrix. The resulting packing was not optimal but an improvement over the original packing. In chapter four we shall show that this type of packing is an NP-complete problem.

In 1974 Domenico Ferrari looked at various clustering methods and found that the nearness matrix method may not be consistent for different reference strings having an exact nearness matrix [Ferrari '74].

Ferrari's basic thesis was to have a relation between the clustering algorithm and the replacement policy. The former aims at improving locality while the replacement policy acts according to some conception of locality as mentioned in the Baeir and Sager
paper. Thus if the clustering methods performs the packing so as to have the locality as near as the replacement policy envisions it. With this aim in mind Ferrari designed his critical working set clustering algorithm which will improve performance if run with the working set policy.

A working set \( W(t,T) \) -- pages referenced in the interval \([t-T+1, t]\) -- becomes a critical working set (CWS) if the next reference \( r_{t+1} \) is not in \( W(t,T) \). And \( r_{t+1} \) is referred to as the critical reference.

The first step of CWS clustering algorithm is to build the CWS matrix which represents the program's blocks. Every time there is a critical reference the entries corresponding to the CWS in the matrix are incremented by one. Once the whole block reference string is read the CWS matrix is complete. A weighted graph is then built with the blocks as nodes and the values in the matrix are added to the edges of the graph. Ferrari then suggests using any of the well known graph clustering algorithms to complete the clustering.

In comparing his results, Ferrari found that the CWS algorithm performed better than the static clustering method of Baer and Caughey, and the dynamic nearness method of Hatfield and Gerald.

For large memories using a larger page size reduces the page fault rate [Joseph '70]. However, clustering techniques may result in a better performance when using small page size. Such clustering method have been suggested by Baer and Sager ['76]. We categorize their method as **dynamic on-line clustering**, because the clustering process is done during execution time. At a page fault requested page
and k-1 neighboring pages are moved to main memory. The requested page is placed at the top of the stack and the other pages would replace the k-1 pages at the bottom of the LRU stack. The replaced pages are moved to the same location in the lower memory. This way least referenced pages are coalesced together and taken away to leave more room for those that might be accessed more often, i.e those near the top of LRU stack. At load time the memory management must check for duplication among those in memory and the pages to be preloaded.

Baer and Sager study showed that dynamic clustering with small page size resulted in significant improvement when large portions of pages (large pages) were unused. In a way this resembles the adaptive preloading scheme.

Over all dynamic clustering avoids the effort and expense of preprocessing the block reference string. However their method has several assumptions and imposes a number of restrictions which make unapplicable for the cache environment:

"...the hardware requirements for use at the level of executable store could not be justified in terms of performance improvements over use of the small page size alone." [Baer and Sager '76].

Another dynamic on-line clustering algorithm has been developed by Bennett and Franaszek['77]. Here the algorithm permutes pages within main memory so as pages that referenced within a given period are moved to locations adjacent to each other, i.e to the same secondary storage block. At a page fault the requested page along with k-1 neighboring pages, those previously clustered together by permutation
into one block, are loaded to main memory.

On experimenting their algorithm on the Advanced Administrative data base, Bennett and Franaszek reported significant improvement as to using block fetching or even the the nearness dynamic clustering.

2.4. Program Transformation

Prefetching algorithms are based on a particular patterns of program behavior, and thus are probabilistic by nature. For that reason they work well for some programs while not as well for other programs. Dynamic clustering algorithms, on the other hand, work on actual program traces. Consequently they tend to work well for all programs. However, the clustering process is long and expensive. An alternative to either method is the program restructuring technique. Here the program itself is transformed so that its set of pages) is reduced and its addressing pattern is more predictable, leading to a higher throughput for multiprogrammed systems [Wolfe '78, Abu-Sufah '79]. The program transformations are usually performed, at the source level [Loveman '77]. Furthermore, many of these transformations have been used to transform scalar programs into vector form for execution on vector machines [Wolfe '78, Kennedy '80]. In this section I shall look at Abu-Sufah's work and study its implications for the cache main memory hierarchy.

Abu-Sufah's objective was to improve program performance in virtual memory systems by reducing the program's working set and forcing
its execution process to go through well defined phases of uninterruptable CPU processing followed by bursts of I/O requests for pages. This goal was accomplished through the use of several transformations on the original program. Some of these transformations had been implemented in the Parafrase compiler [Wolfe '78]; others were new. These transformations have a significant effect on programs that process large aggregate data arrays. The processing of these arrays is always done within program loops; thus Abu-Sufah's transformations are for these loops.

First, Abu-Sufah labels the ideal program model as one that goes through alternating periods of clustered I/O requests and CPU activities. The type of loops that follow this model are called elementary loops, they are defined as:

"An elementary loop is an ordered set of assignment statements preceded by one Do control statement. The variables referenced in the loop are one dimensional arrays and possibly scalars. The subscripts of the array variables are linear functions of the index variable. In the subscript expressions, all the index variables have the same coefficient."

The ideal program model is then referred to as the elementary loop model (ELM). An elementary loop could be modeled by the ELM.

Abu-Sufah introduced four loop transformations that would reduce the memory requirement of loops modeled by the ELM. The application of any of these transformations is always preceded by a dependency analysis.
Two statements in a program are either not related or have one of the three types of dependencies between them: true dependence where the output of the first statement is used in the second statement, antidependence where a variable used in the first statement is redefined in the second statement, or output dependence where the variable is defined in both statements.

Dependency analysis entails the discovery, representation, and the breaking of the dependencies among the statements of a loop. First a dependency graph is built with the set of nodes representing the set of statements for the program. A direct edge is drawn between any two dependent statements. Each edge would have a value of: zero, one, or two indicating true dependence, antidependence, or output dependence respectively.

Statements within a graph cycle constitute a recurrence, a group of statement that all depend on each other. These statements cannot be executed in parallel. Thus the second phase of the dependency analysis involve the discovery and breaking of cycles. Removing any essential dependency within the cycle breaks the cycle.

Several techniques have been introduced in the Parafrase compiler to break dependencies. These include scalar transformations such as scalar renaming, induction variable substitution, and forward substitution to scalars used in subscript expressions [Wolfe '78, Abu-Sufah '79]. To these techniques, Abu-Sufah adds scalar expansion to array variables, and forward substitution. Two of the transformations proposed by Abu-Sufah are applicable to our current problem;
they are: horizontal and vertical distribution.

**Horizontal distribution** is the subdivision of the original loop into several loops each having one statement. This is the simplest of the four transformations, and was first introduced by Kuck [Kuck '78], but is refined by Abu-Sufah to be applied to loops of ELM. It is only applicable after dependency analysis is done on the given loop and all recurrences are removed. Other dependencies found in the original loop are preserved across the new loops. A vectorizer would make each of the new loops a vector operation.

**Example**

```plaintext
DO 10 I = 1, N
S1  A(I) = A(I) - C(I)
S2  C(I) = C(I)*B(I) - 2
S3  B(I) = B(I)*C(I) + B(I)
10 CONTINUE
```

becomes

```plaintext
DO S1 I = 1, N
S1  A(I) = A(I) - C(I)
DO S2 I = 1, N
S2  C(I) = C(I)*B(I) - 2
DO S3 I = 1, N
S3  B(I) = B(I)*C(I) + B(I)    []
```

For a given loop if none of the statements included all of the arrays referenced in the loop, then this transformation reduces the working set size for the loop. However, the I/O activity for the loop
increases. Pages that could have been accessed more than once while in memory for the original loop must be replaced and requested with the new structure. Similarly block traffic between the cache and main memory would increase leading to a higher cache miss ratio. Moreover, the CPU processing time will increase due to the execution of the extra control instructions of the new loops.

**Vertical distribution** is a strip mined version of the horizontal distribution. Here additional control statements, called the page indexing statements, are introduced to allow the processing of one page per array of the original loop. The problems of increased I/O transfers introduced by the horizontal distribution and the scalar expansion of array variables are partially eliminated by this transformation.

**Example**

```
DO 10 I = 1, N
S1  T = A(I)*C(I)                S1
S2  A(I) = D(I) + B(I)           S2
S3  B(I) = T + F(I)/E(I)         S3
10  CONTINUE
```

In this loop statement S2 is antidependent on S1 and is executed before S3. Thus S1 could not be forward substituted to S3. For this reason the scalar expansion to array variable transformation must be used to allow for the application of the horizontal distribution transformation:
DO S1 I = 1, N
S1 \[ T'(I) = A(I) \times C(I) \]

DO S2 I = 1, N
S2 \[ A(I) = D(I) + B(I) \]

DO S3 I = 1, N
S3 \[ B(I) = T'(I) + F(I)/E(I) \]

Let Z be the page size in words and let '/' be an integer division i.e the result of the division is an integer. Then the vertical distribution is:

DO 10 IP = 1, N/Z
ILB = 1 + (IP-1)\times Z
IUB = MIN(IP \times Z, N)

DO S1 I = ILB, IUB
S1 \[ T'(I \mod(Z) +1) = A(I) \times C(I) \]

DO S2 I = ILB, IUB
S2 \[ A(I) = D(I) + B(I) \]

DO S3 I = ILB, IUB
S3 \[ B(I) = T'(I \mod(Z) +1) + F(I)/E(I) \]
10 CONTINUE

The values of ILB and IUB make sure that the arrays do not cross page boundaries and cause page faults. They are the page indices. The index variable for the array T' insures that we use only one page for the whole array T'. In other words the page for T' need not copied to secondary storage. Here Abu-Sufah simulates our idea of a temporary cache block Pages A in statement S1 and B in S2 are accessed in S2 and S3 respectively, before being moved to the secondary store; thus reducing the I/O transfers. The memory requirement is the same as
that produced by the horizontal transformation. This puts the vertical distribution at the top of the list of loop transformations given by Abu-Sufah.

Compared to horizontal distribution the vertical distribution may increase the block hit ratio and reduce the block traffic, leading to a lower miss ratio. The execution time for the page indexing statements and those evaluating ILB and IUB may offset the gain of lower miss ratio. A detailed analysis for this possibility is presented in the following section. Our immediate objective is to show how the cache allocation models could further enhance the vertical distribution transformation.

First the compiler need not allocate a page for the temporary array like T' but rather allocate a temporary cache block; thus further reducing the memory requirement for loop. Moreover, the allocation method for temporary blocks eliminates the need for the index variable for arrays like T' as suggested by Abu-Sufah, but rather use the same index variable used by the other arrays in the loop.
CHAPTER 3

Two Models For Cache Management By The Compiler

3.1. Introduction

In the previous chapter we presented a survey of literature on the significance of page clustering (packing), prefetching, and program transformation in reducing the page fault rate and subsequently improving the system's performance. In this chapter we shall put to use all the major concepts of the three enhancement methods, but in a form germane to the cache/main memory hierarchy. This task is accomplished using two different cache management models.

Many of the performance enhancement models (prefetching or clustering) are based on the static analysis of the program's data flow and addressing pattern. Nevertheless, dynamic program behavior may be different. The model which proves to effectively enhance the systems' performance under various execution conditions is selected for implementation. The SL prefetching algorithm by Baier and Sager ['76] is a good example.

When suggesting a model, designers presume a number of possible program behaviors, then study their effects under a particular computer architecture. The model is then built to improve the system's performance by making use of the knowledge gained from the study. For example, when designers introduced the prefetching models they assumed that programs exhibit spatial localities, and that the access
time for secondary storage is much larger than a single page transfer
time. With such a premise all preloading, in the prefetching models,
are done at a page fault. The page causing the fault and the page
next to it in the address space are both loaded to main memory. This
leads to lower page faults and better system utilization.

The employment of the prefetching models for the cache memory
hierarchy may not result in lower cache misses or any performance
gain simply because many of the undertaken assumptions considered in
the construction of these models no longer hold. In particular, the
access time and transfer time are indistinguishable in the cache main
memory hierarchy. As such, preloading at a cache miss, if not clev-
erly implemented, requires twice the time as serving a single cache
miss i.e no performance gain. Moreover, a wrong prediction in such a
preloading scheme may double the cache miss penalty.

A possible alternative would be to preload during program execu-
tion, using some form of cycle stealing while the cache main memory
bus is free. In other words, cache loading and program computations
could be performed in parallel. When a cache miss occurs, preloading
must be aborted to release the memory bus for serving the cache miss,
so as not to increase the miss penalty.

One way of preventing a conflict between a preload and a miss is
to preload during the execution of a cycle (a loop), when a miss is
less likely to occur and the cache main memory bus is free. This
makes cycles and cycle detection vital for cache prefetching under
dynamic allocation. Loop transformations presented in the previous
chapter will prove useful for block prefetching schemes. Examples of these concepts are presented later in this chapter.

3.2. Data Classification and Packing

Rather than mimicking the prefetching and clustering models as presented in the previous chapters, we decided to take a different look at ways for improving cache performance. Our objective is to lower the block traffic between the cache and main memory. In essence, the design reduces the cache miss ratio, the penalty time, and diminishes the effective cycle time. This is accomplished through a number of processing stages.

First, we classify four types of cache blocks depending on the data within them. Thereafter we attribute a cost of transfer with each of the four types. Our models will thereupon try to minimize the total cost of transfers between main memory and the cache.

For both models the data of the given program is classified into one of four types then packed to an appropriate block. Thereon, each model will manage these blocks so as to reduce the transfer cost.

There are four types of variables:

1) Temporaries ($V_t$),
2) Constants ($C$),
3) Input variables ($V_i$),
4) Computed variables ($V_c$).

For these variables, a conventional compiler will do two types of allocation: storage allocation and register assignment. In this chapter we are proposing a new function: cache management.
In our design, the compiler classifies the program variables into one of the four types. It subsequently builds a conflict graph for each class. Variables of the same type that are in conflict are packed in the same block. If the number of variables that are in conflict exceeds the size of a block, then various heuristic techniques are used to determine the packing order.

Once packing is complete, a table for the blocks is built. The entries in the table give the block number, the block type, and the address of the first variable in the block, i.e., the address of the block. Thereafter the table is passed to the cache allocation models, which try to cut down the block traffic cost on the cache main memory bus.

Each time a cache block uses the cache main memory bus, on load or replacement, the cost of the program increases by one. A cache management scheme needs to schedule the loading and replacement of these blocks in order to minimize the total cost of running the program. For example, traditional cache management depends on good replacement policies which reduce a program's total cost by not replacing blocks that might be accessed again, since such replacement increases the total cost by one or more; one for non-modified blocks and two or more for modified blocks, since these blocks have to be copied to main memory while the non-modified blocks are simply written over. On the other hand, our cache management models will be doing both loading (preloading) and replacement. The aim here is to reduce block traffic cost between the cache and main memory.
To formalize the previous concepts, we attribute each cache block with a birth point, a death point and a life span. The birth point occurs when the block is first accessed. A death point occurs when the program no longer accesses that block. The life span of a block is then the interval of time between its birth and its death. Block packing will have a direct affect on the three values.

**TABLE 1**

<table>
<thead>
<tr>
<th>Block Type</th>
<th>C</th>
<th>V_t</th>
<th>V_r</th>
<th>V_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birth point cost</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Death point cost</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Premature replacement cost</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

With each cache block we attribute a cost at its birth point and at its death point depending on its type (table 1). Blocks of types C and V_i must first be loaded to the cache before their use and thus have a cost of one at their birth point. On the other hand, blocks of types V_c and V_t need not be brought to the cache to be modified. Instead, a block in the cache can be selected to hold the newly defined or temporary variable. Thus blocks of types V_c and V_t have a zero cost at their birth point.

At their death point blocks of types C and V_t are simply overwritten for replacement and thus have a zero cost. However, the modified blocks of V_c and V_i have to be copied to main memory at their death point. This means that both V_c and V_i have a cost of one at their death point.
The replacement of a block before its death point is a premature replacement (table 1). Premature replacements are costly and a good replacement policy should replace the blocks that would contribute least to the total cost.

For example, the cost of premature replacement of a block of temporaries \( V_c \) is two: one for writing it to main memory and one for reading it back. Similarly for blocks of \( V_c \) and \( V_t \). A premature replacement of C blocks has a cost of one since they could be overwritten without a write back, but must be read later.

In the following example we show how Belady's Min algorithm [Belady '66] makes the fewest premature replacements but not necessarily those replacements which contribute the least to the total cost.

**Example**

Given the following block reference string and a cache with three blocks:

\[
C V_t V_c V_c2 C V_t V_c1
\]

Belady's optimal algorithm:

\[
C C C C C C C \\
0 V_t V_t V_t V_t V_t V_t V_t \\
0 0 V_c V_c2 V_c2 V_c2 V_c1
\]

| Partial cost | 1 | 1 | 1 | 2 | 2 | 2 | 4 |

Notice here that \( V_c1 \) was the first block to be replaced but since it will be accessed again it contributed to a cost of two: one for
writing it to main memory and one for reading it back. The resulting
total cost is 5 since $V_{cl}$ will be written again to main memory.

A less costly scheme would be:

```
c  c  c  v_{c2}  c  c  c
0  v_{tl}  v_{tl}  v_{tl}  v_{tl}  v_{tl}
0  0  v_{cl}  v_{cl}  v_{cl}  v_{cl}  v_{cl}
```

Partial cost 1 1 1 1 3 3 3

The total cost is 4 since $V_{cl}$ has to be written back to main
memory. The decision here was to replace C because it had the least
replacement value of one, rather than replacing $V_{cl}$ which has a
premature replacement of two even though it is further away from
being accessed. []

Attributing a value for each possible replacement was first stu-
died by and [Horowitz et. al.] [Kennedy '71]. In his thesis, Ken-
nedy looked at various ways for optimally allocating and dealloca-
ting index registers for the IBM 7090.

If blocks have a high life span, then there will be a tendency
for the cache to be congested which might result in several premature
replacements. On the other hand, if blocks have a low life span then
the possibility of premature replacement diminishes drastically.
Packing has a direct effect on the life span of a block. Thus, when
doing the packing, considerations for reducing the life span must be
taken into account.
Another way for reducing premature replacements would be by scheduling the assignment and the replacement of blocks. These tasks are performed by the management models that are presented in the following sections.

3.3. Cache design requirements

Originally each cache block has two tag bits: the validity bit, and the modification bit. Both of these bits are hardware controlled.

The validity bit is set to one when a block of datum is first brought from main memory. This indicates that the information within the block is valid for use by the running program. The validity bit is reset to zero when the whole cache is purged (invalidated). This happens when there is a hardware malfunction, when the system is first powered on (booted), or when a corresponding main memory address is modified by an I/O channel. Some systems invalidate the cache at task switching. Such measures are taken to prevent processes from accessing data of a previous process.

When the CPU writes into a block, the modification bit is set to one. If a block selected for replacement has its modification bit set to one, then this block is copied to main memory first before the requested block could overwrite it.

In a write-through policy, the modification bit is not used (or required) since the corresponding main memory location is updated each time there is a write to a block. Our assumption here that a write-in policy is being used.
In conventional compilers there is no distinction for the storage of variables. This may lead to a mixing among all of the variables in the cache blocks which may augment the number of modified blocks. Consequently, block traffic would increase in the cache main memory bus. Thus variable separation may effectively cut down the number of modified blocks. Furthermore, not all modified blocks need to be copied back to main memory. For example, blocks holding temporaries need not be copied back but simply overwritten if not needed. Nonetheless, if they are needed and have to be replaced then they must be saved in main memory and brought back to the cache as soon as possible. However, blocks of types $V_c$ and $V_i$ have to be copied to main memory at replacement.

To facilitate the distinction between block types and their requirements, we introduce two indicator bits to each cache block: the CPU useful bit and the memory useful bit. These bits are software controlled by the pseudo-instructions introduced by the cache allocation models.

When the CPU useful bit is set to one it signifies that the data within the block is live and will be accessed by the CPU. Blocks having their CPU useful bit set to zero are the first to be replaced. If however, a block with a one CPU useful bit is replaced then it must be brought to the cache as soon as possible.

When the memory useful bit is set to one it indicates that the data within the block is needed by main memory. At replacement, modified blocks that have their memory useful bit set to one are copied
to main memory while the other modified blocks are simply overwrit-
ten. A precondition for overwriting a cache block without a copy back
is that both of its bits (CPU and memory useful) are equal to zero.

To facilitate prefetching we shall require another indicator
bit: lock bit. Originally all the lock bits are set to zero. When a
block is requested by a prefetch pseudo-instruction the cache line
(block) selected for replacement will have its lock bit set to one.
The bit is reset to zero when the requested data is loaded. Compu-
tation and prefetching could proceed in parallel. A wait only occurs if
a block with lock one is accessed. Computation would resume only when
the memory transfer for that block is complete i.e the requested data
is loaded.

In addition to the three indicator bits, a prefetch register is
required where the address of the data to be preloaded is placed.
Smith [’78] reports that the Amdahl 460 V6 computer has such a regis-
ter.

The use of the prefetch register and the tag bits will be con-
trolled by pseudo-instructions that are presented as part of the
allocation models given in the following section.

2.4. The Management Models

The two models are called the prompting model and the explicit
management model. In the former requests for cache blocks are
prompted before they are actually needed with hope of averting a
cache miss when the data is requested by the CPU. If, however, the
data is not in the cache when needed, then a cache miss occurs. In this model, the replacement

For the explicit model allocation and deallocation of cache blocks is controlled by the pseudo-code placed there by the compiler. In essence the pseudo-code resembles register load and store instructions that are produced by the explicit allocation part of the code generator for the compiler.

The advantage of the explicit model is that the compiler knows the status of every block in the cache. A basic assumption for this model is that only one process is using the cache at any time. If any other process uses the cache and replaces live blocks then these blocks must be brought back to the cache before the previous process is initiated. This is similar to the register save operation during the serving of an interrupt. Another possibility would be to allow the other process to use blocks that are not being used by the current process, or simply allow each process to have a 'working set' of cache blocks. A list of such blocks could be provided by the compiler to the operating system. Such a scheme would function well in a fully associative cache, since any address could be mapped to any block making any free block useful for storage.

Measurements done by Peuto and Shustek['77] showed that user-initiated supervisor calls (SVCs) cause a high number of premature replacements of the user's cache blocks:

"...we simulated the cache activity for one job with large number of SVCs first assuming a 100% cache flush for each SVC, and then again with no flush; the number of misses
changed by a factor of 10. Measurements showed that the actual fraction of the cache displaced by an SVC varies from 0.16 to 1.0 and that all non-trivial requests completely replace the cache."

The implementation of the explicit model is thus vital in reducing the effects of SVCs by allowing these requests to use free cache blocks first and displace blocks of least cost.

Various heuristics for the packing and the scheduling of blocks will be covered in the fifth chapter. In this chapter we will present and discuss the pseudo-code for each model.

3.4.1. The prompting model

In this model there are four types of pseudo-instructions represented by two mnemonics: prompt and release. The prompt instruction has three operands while the release instruction has two operands. For both instructions the first operand is the main memory address of the block (i.e. address of the first variable in the block), the second operand indicates the type of operation to be performed. The type of operation is represented by a zero or a one. The third operand in the prompt instruction gives the setting of the memory useful bit. The instruction format will look like:

```
memory     operation     memory
Prompt { block , type , useful bit }
     address     bit     setting
```

```
memory     write
Release { block , back }
     address     bit
```
For the prompt instruction, a one in the second operand means that the address given in the first operand must be fetched from main memory. This is done by placing the given address in the prefetch register. The block chosen to hold the incoming data will have its lock bit set to one. Such operations are used for fetching the blocks of constants, and the blocks of input variables.

On the other hand, temporaries and computed variables are allocated by a prompt instruction whose second operand is zero. In this case a free block or a block with lowest cost is chosen (replaced), and the value of its descriptor is changed so as to refer to the address in the first operand of the given prompt instruction.

In summary the prompt instruction manages the allocation part of this model, while the release instruction is concerned with deallocation of cache blocks.

The deallocation for the release instruction is done by setting the CPU-useful to zero. A value of zero in the second operand of the release instruction means that the block is of no use to this process and it should be placed at the top of the list of free blocks. On the other hand, a value of one would require that the block be copied back to main memory as soon as possible. Once copy back is successful the memory-useful bit of the block is set to zero. This operation is done for computed and input blocks. Any block having both of its usefulness bits set to zero is a free block.
Example

Assume we have a cache with four blocks. Let the block size be 3 words. Given the following sequence of statements:

\[ \begin{align*}
E &= A*B + 3*D \\
F &= A*B + 6 \\
G &= 5*D \\
\end{align*} \]

The quadruples are:

\[ \begin{align*}
t1 &= A*B \\
t2 &= 3*D \\
t3 &= t1 + t2 \\
E &= t3 \\
t4 &= A*B \\
t5 &= t4 + 6 \\
F &= t5 \\
t6 &= 5*D \\
G &= t6 \\
\end{align*} \]

The corresponding cache blocks are:

\[ \begin{align*}
V_{t1} &= \{t1, t2, t3\} \\
V_{t2} &= \{t4, t5, t6\} \\
C1 &= \{A, B, D\} \\
V &= \{E, F, G\} \\
C2 &= \{3, 5, 6\} \\
\end{align*} \]
The prompting model would add the following pseudo-code:

Prompt \((C1,1,0)\)  
do a memory read and set memory

Prompt \((C2,1,0)\)  
useful bit to zero  
i.e read only block

Prompt \((V_{c1},0,0)\)  
select a block i.e no read and memory useful bit is zero

\[ t1 := A \times B \]
\[ t2 := 3 \times D \]
\[ t3 := t1 + t2 \]

Prompt \((V_{c1},0,1)\)  
select a block, set memory  
useful bit to one

\[ E := t3 \]

Release \((V_{t1},0)\)  
Throw away this block

Prompt \((V_{t2},0,0)\)

\[ t4 := A \times B \]
\[ t5 := t4 + 6 \]
\[ F := t5 \]
\[ t6 := 5 \times D \]
\[ G := t6 \]

Release \((V_{c1},1)\)  
Copy back this block to main memory

Release \((V_{t2},0)\)
Release \((C1,1,0)\)  
Throw away these blocks
Release \((C2,0)\)  
[]

In the previous chapter we talked about the usefulness of the vertical distribution for our model. In what follows we shall show that while executing one of the inner loops we could prompt for blocks of the following inner loop.
Example

Let R be the block size and '/' be an integer division. Using the prompting model on the loop given in the previous chapter, we get:

```
Prompt (A(1),1,1)       Here array A is modifiable
Prompt (C(1),1,0)       Here array C is read only
         ....
         ....
         ....
         ....
         ....
         ....
         ....
DO 10 IP = 1, N/R - 1
ILB  = 1 + (IP-1)*R
IUB  = IP*R
Prompt (T'(ILB),0,0)
Prompt (D(ILB),1,0)
Prompt (B(ILB),1,1)
DO S1 I = ILB, IUB
T'(I) = A(I)*C(I)
S1 CONTINUE
Prompt (F(ILB),1,0)
Prompt (E(ILB),1,0)
DO S2 I = ILB, IUB
A(I) = D(I) + B(I)
S2 CONTINUE
Release (A(ILB),1)
Prompt (A(IUB+1),1,1)
Prompt (C(IUB+1),1,0)
DO S3 I = ILB, IUB
B(I) = T'(I) + F(I)/E(I)
S3 CONTINUE
Release (B(ILB),1)
Release (T'(ILB),0)
10 CONTINUE
ILB  = 1 + (IP-1)*R
IUB  = N
Prompt (T'(ILB),0,0)
Prompt (D(ILB),1,0)
Prompt (B(ILB),1,1)
DO S1 I = ILB, IUB
T'(I) = A(I)*C(I)
S1 CONTINUE
Prompt (F(ILB),1,0)
Prompt (E(ILB),1,0)
DO S2 I = ILB, IUB
A(I) = D(I) + B(I)
S2 CONTINUE
```
Release (A(ILB),1) Notice that we avoided prompting for A and C here
DO 3 I = ILB, IUB
   B(I) = T'(I) + F(I)/E(I)
3 CONTINUE
Release (B(ILB),1)
Release (T'(ILB),0) []

In the previous example we demonstrated how the prompting model when used in conjunction with the vertical distribution could eliminate all the cache misses and the time for block write back. A point of interest is the gain or loss in CPU time as a result of this transformation. This will be investigated in the next chapter.

3.4.2. The Explicit Management Model

In the prompting model the compiler places a request for a load within the code. The decision of when to do the actual load is completely controlled by the hardware. Moreover, the compiler instructs the hardware for blocks that are needed for temporaries and computed variables; the time of assignment or allocation is also controlled by the hardware. And furthermore, the compiler suggests which blocks are free or need to be copied back to the main memory. Nevertheless, the hardware has the full responsibility for the replacement policy. Thus the prompting model aids the hardware in achieving a lower miss ratio and penalty time. In return an error in the replacement or prefetching is tolerated by satisfying the need of cache miss and will not cause a program error.

In the explicit management model, software takes the full responsibility for allocating and deallocating the cache. A premature
replacement, which is not followed by an immediate preload request for the replaced block, may result in a program error. Cache loading is done at the time of the request with no delay. A write-back request to main memory could be delayed if it will not result in a premature replacement. Any incoming data block would overwrite the existing data regardless of its type (memory useful or CPU useful).

The software of this model has a full knowledge of the status of each block in the cache. Hence the tag bits: memory useful and CPU useful, are no longer needed. In spite of that, these tag bits may be simulated at compile time to aid in the allocation and replacement decisions.

Under a static allocation scheme and with the knowledge of the cache mapping function, the compiler would be able to distinguish where each block of data resides in the cache. Combining this information with the scheduling algorithm makeup the major design of this model. The details of the block packing and the scheduling algorithms are the topic of the fifth chapter.
The explicit management model has four types of pseudo-instructions: load, assign, and transfer. Each instruction has two operands. The first operand gives the address of the cache block concerned. The second operand represents the main memory address of the block i.e. the address of the first variable in the block:

\[
\begin{array}{ll}
\text{Load} & (\text{block } , \text{ block } ) \\
\text{Assign} & (\text{block } , \text{ block } ) \\
\text{Transfer} & (\text{block } , \text{ block } )
\end{array}
\]

In executing the load instruction the address given in the second operand is requested from main memory and then placed in the cache block indicated by the value of the first operand. The block selected in the load instruction will have its lock bit set to one. The main memory address requested is placed in the prefetch register. Cache loading and computation could proceed in parallel. Computation halts only when there is an attempt to access a block whose lock bit is one. In this case the previous load instruction resembles the cache miss, it is simply a 'software' cache miss. The rationale here is that the value requested will be used for computation and if it is not in the cache then it would cause a cache miss. However, cache misses are not tolerated in the explicit model and a missing value may result in a program error. It is just like accessing a CPU register for a value that has yet to be loaded. The execution time for the load instruction is equal to one cache cycle plus the number of cycle
that might occur in case of a wait. This instruction is usually used for the loading of blocks of constants (C), and input variables (\(V_i\)). Moreover, if any live block is prematurely replaced from the cache then it is immediately reloaded using the load instruction.

The compiler objective would be to place load request before the actual access of the data, in such a way as to minimize the waiting time. Thus execution of the program and cache loading could proceed in parallel.

The `assign` instruction allocates the cache block given in the first operand to the variables whose address is given in the second operand. The execution time for this instruction requires only one cache cycle time. This instruction is used for the \(V_t\) and \(V_c\) type blocks.

The `transfer` instruction writes the cache block indicated by the first operand to the main memory address given in the second operand. The execution time of this instruction is one cache cycle. Any load instruction must wait for the completion of any previous transfer instruction before being executed since the cache main memory bus is busy serving the transfer instruction. Generally, the transfer instruction is used for writing back the \(V_c\) and the \(V_i\) type cache blocks.
Once a cache block is allocated all addressing requests will be directed towards that block. In other words, the CPU will generate only cache addresses for data needed. Each address will have the form:

\[ \text{Cache_block_number(Offset\_within\_block)} \]

The block number is designated by the load or assign pseudo-instructions and the offset is determined after the block packing is complete. Both of these values are computed at compile time. In consequence the effective address calculation time is minimized. Access to main memory is only allowed by the load pseudo-instruction.
**Example**

From the previous example we have:

\[ V_{t1} = \{t1, t2, t3\} \quad V_{t2} = \{t4, t5, t6\} \quad C1 = \{A, B, D\} \]

\[ V_c = \{E, F, G\} \quad C2 = \{3, 5, 6\} \]

<table>
<thead>
<tr>
<th>Load</th>
<th>(C1,B1)</th>
<th>Load block C1 from main memory and place in block 1 of the cache.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>(C2,B2)</td>
<td>Load block C2 from main memory and place in block 2 of the cache.</td>
</tr>
<tr>
<td>Assign</td>
<td>(V_{t1}, B3)</td>
<td>Allocate cache block 3 to ( V_{t1} ) without a memory load.</td>
</tr>
<tr>
<td>Assign</td>
<td>(V_c, B4)</td>
<td>Allocate cache block 4 to ( V_c ) block of variables to be computed.</td>
</tr>
<tr>
<td>LD</td>
<td>1,B1(0)</td>
<td>( t1 := A \times B )</td>
</tr>
<tr>
<td>LD</td>
<td>2,B1(1)</td>
<td>( t2 := 3 \times D )</td>
</tr>
<tr>
<td>MR</td>
<td>1,2</td>
<td>( t3 := t1 + t2 )</td>
</tr>
<tr>
<td>ST</td>
<td>1,B3(0)</td>
<td>( E := t3 )</td>
</tr>
<tr>
<td>LD</td>
<td>1,B2(0)</td>
<td>( t2 := 3 \times D )</td>
</tr>
<tr>
<td>LD</td>
<td>2,B1(1)</td>
<td>( t2 := 3 \times D )</td>
</tr>
<tr>
<td>MR</td>
<td>1,2</td>
<td>( t3 := t1 + t2 )</td>
</tr>
<tr>
<td>ST</td>
<td>1,B3(1)</td>
<td>( E := t3 )</td>
</tr>
<tr>
<td>LD</td>
<td>1,B3(0)</td>
<td>( t3 := t1 + t2 )</td>
</tr>
<tr>
<td>LD</td>
<td>2,B3(1)</td>
<td>( t3 := t1 + t2 )</td>
</tr>
<tr>
<td>AD</td>
<td>1,2</td>
<td>( t3 := t1 + t2 )</td>
</tr>
<tr>
<td>ST</td>
<td>1,B3(2)</td>
<td>( E := t3 )</td>
</tr>
<tr>
<td>LD</td>
<td>1,B3(2)</td>
<td>( E := t3 )</td>
</tr>
<tr>
<td>ST</td>
<td>1,B4(0)</td>
<td>( E := t3 )</td>
</tr>
</tbody>
</table>
Assign \((v_2, b_3)\)  
Allocate block 3 to the \(v_2\) block

LD 1.B1(0)  
t4 := A * B

LD 2.B1(1)  

MR 1,2  

ST 1.B3(0)  

LD 1.B3(0)  
t5 := t4 + 6

LD 2.B2(2)  

AD 1,2  

ST 1.B3(1)  

LD 1.B3(1)  

ST 1.B4(1)  

F := t5

LD 1.B2(1)  

ST 1.B4(1)  

LD 2.B1(2)  

MR 1,2  

ST 1.B3(2)  

LD 1.B3(2)  

ST 1.B4(2)  

G := t6

Transfer \((v_1, 4)\)  
Write block 4 to main memory

An important function in the explicit model is the storage management of aggregate data elements; arrays in particular. On this account we shall introduce a new indexing scheme which will facilitate the handling of large array elements in the cache. This is accomplished by using a special index register, and the application of three additional pseudo-instructions: **group**, **loadind**, and **transferind**.

At compile time each array variable is allocated a fixed number \((n)\) of contiguous cache blocks. The actual block assignment is then carried out by executing the group pseudo-instruction:
cache number of
Group (block , blocks )
number allocated

The cache block number is the number of the first block in the
group. Access to the array, or a block prefetch will be through these
assigned (n) cache blocks.

For each array Ai the compiler allocates Mi main memory blocks,
these blocks will be mapped to the n blocks allocated for the array
in the cache. The congruence mapping function is employed here. In
other words, as far as the array is concerned the cache is a direct
mapping cache of size n.

Main memory load (preload) requests for an array are initiated
by the loadind instruction. This instruction has three operands. The
first operand is the first block in the group of blocks assigned to
the array in the cache by the group instruction. The second operand
is the first block in the group of blocks assigned to the array in
main memory. The third operand is the index value generated by the
program for accessing the array elements. The value of i is usually a
one, or multiple of m, the block size.

Executing the loadind(B1,M2,i) will cause the main memory block
number: M2 + |i/m| to be prefetched and loaded to the cache block
number: B1 + |i/m| mod n.
In this model a sequence of instructions might look as follows:

```
Group   (Bl,n)
......
LoadInd (Bl,M2,1)
......
LD     l,Bl(f)
```

Arrays of type \( V_c \) or \( V_i \) are copied back to main memory by executing the transferInd instruction. This instruction uses the same three operands of the loadInd instruction, but for write-back rather than reading from main memory.

We shall look at the register model in a loop construct that has been restructured using the vertical distribution.
Example

Let R be the block size and '/=' be an integer division. Using the register model the loop of the previous example will be:

\[ V_c = (IP, ILB, I) \quad C = (N, R, -) \]

Assign \((V_c, B_0)\) allocate \(B_0\) for computed scalars
Load \((C, B_1)\) allocate the constants to \(B_1\)
Group \((B_2, 1)\) \(B_2\) will be used for \(T'\).
Group \((B_4, 2)\)
Group \((B_6, 2)\)
Group \((B_8, 2)\)
Group \((B_{10}, 2)\)
Group \((B_{12}, 2)\)
Group \((B_{14}, 2)\)

DO 10 IP = 1, N/R
   ILB = 1 + (IP-1)*R
   IUB = IP*R
   Loadind \((B_4, A, ILB)\) prefetching array \(A\).
   Loadind \((B_6, C, ILB)\) Array \(C\) loaded to block \(B_6\)
   Loadind \((B_8, D, ILB)\) Array \(D\) is placed in block 3
   Loadind \((B_{10}, B, ILB)\)

DO S1 I = ILB, IUB
   B2(I) = B4(I)*B6(I)
S1 CONTINUE

   Loadind \((B_{12}, F, ILB)\)
   Loadind \((B_{14}, E, ILB)\)

DO S2 I = ILB, IUB
   B4(I) = B8(I) + B_{10}(I)
S2 CONTINUE

Transferind \((B_4, A, ILB)\) This cache block holds array \(A\)

DO S3 I = ILB, IUB
   B_{10}(I) = B2(I) + B_{12}(I) / B_{14}(I)
S3 CONTINUE

Transferind \((B_{10}, B, ILB)\) Copy back blocks for array \(B\)

10 CONTINUE
In the previous example we demonstrated how the explicit model when used in conjunction with the vertical distribution could eliminate most of the block load time and the time for the block write back. In addition, we used only one block for the temporary array T'.

3.5. Conclusion

In this chapter we touched on several topics that are essential for the implementation of the two management models. An exhaustive investigation of all these topics will not be covered in this thesis, rather we shall be content with a thorough theoretical analysis of packing, and the conditions for doing the various transformations.
CHAPTER 4
Analysis of Improvement Methods

4.1. Introduction

After looking at various cache designs, preloading methods and clustering techniques, we shall formalize the importance of these concepts for cache performance. Moreover, we would like to study the affects of the various improvements methods on each other.

In the first chapter we chose the cache miss ratio as our performance measure. Although we might have made a reasonable choice, many designers prefer to have the percentage of execution time lost in serving the cache misses as a performance parameter. We shall call this value: the CPU miss idle time, or simply $I_c$. This new parameter is formulated in the following lemma.

Lemma 4.1

The CPU miss idle time is:

$$I_c = S \times R_i \times (T_p / T_i)$$

Where

$S$ is the cache miss ratio

$R_i$ is the average number of memory references per instruction

$T_p$ is the cache miss penalty time
$T_i$ is the average execution time for an instruction assuming an infinite cache.

**Proof**

The CPU idle time, the value of $I_c$, is simply the ratio of the time spent serving the cache misses ($C_p$) to the total execution time ($E_t$) assuming no cache misses. The value of $E_t$ is computed by taking the product of the average execution time for each instruction ($T_i$) and the number of instructions executed ($N_i$). While the value of $C_p$ is the result of the product of the number of cache misses ($N_m$) and the cache miss penalty ($T_p$).

$$I_c = \frac{N_m \times T_p}{N_i \times T_i}$$

Let $R_i$ be the average number of memory references per instruction. The total number of memory references ($M_r$) is then equal to:

$$M_r = R_i \times N_i$$

In other words: $N_i = M_r / R_i$. And therefore $I_c$ becomes:

$$I_c = \frac{N_m}{M_r} \times R_i \times \frac{T_p}{T_i}$$

But by definition the cache miss ratio ($S$) is the ratio of the number of cache misses to the total number of memory references; thus $I_c$ becomes:

$$I_c = S \times R_i \times \frac{T_p}{T_i}$$

[Q.E.D]

The IBM 370 offers three instruction lengths: two, four, and six bytes. The two byte instructions usually require one memory reference.
i.e. for accessing the instruction itself. The four and six byte instructions make two or more memory references.

In a benchmark of seven programs, in which more than a hundred million instructions were executed, the average length of the instructions executed was about 3.6 bytes [Peuto and Shustek '77]. The four-byte instructions made up more than 70% of the total number of instructions executed. Thus it is safe to assume that on the average there are two memory references per instruction.

Therefore, for an IBM 370 architecture:

\[ I_c = 2 S \times (T_i/T_p) \]

Example:

Here we look at two computers having the same instruction set but different cache structures, they are the IBM 370/168 and the Amdahl 470 V6. The former has a 32K byte four way set associative cache and a write-through writing policy while the later has a 16K byte two way set associative cache with the write-in policy.

The values of \( T_p \) for the IBM 370/168 and the Amdahl 470 are 480ns and 650ns respectively. On the other hand, the average instruction times are 300ns and 160ns respectively [Peuto and Shustek '77]. The value of \( I_c \) is then (3.2 \times S) for the IBM machine and (8.2 \times S) for the Amdahl machine. This makes the Amdahl 470 extremely sensitive to the cache miss ratio. []

The previous example illustrates the importance of the cache miss ratio in high performance computers. Moreover, the value of
\( \frac{T_p}{T_1} \) reflects the speed gap between the CPU and main memory. Thus an increase in the speed gap, due to an increase in the cache miss penalty, would subsequently increase the CPU idle time. As I pointed out earlier in the thesis, we may expect this mismatch to continue to grow because it is economically inconceivable to have large memories at speeds comparable to the CPU.

Generally the cache miss penalty is assumed to be hardware dependent. Nevertheless, the cache miss penalty could be masked with the CPU execution time. This has been demonstrated in the previous chapter.

We have seen in chapter two of this thesis how the preloading and packing (clustering) could reduce the page fault rate. We also suggested that both of these methods could also be used to reduce the cache miss ratio. Thus, there must be a dependence relation between these improvement methods and the miss ratio. In 1978 Hayes presented, without a formal proof, such a relation [Hayes '78]; in this section we provide a formal proof of Hayes's formula.

**Definition**

The **packing factor** is the average number of consecutive memory references to the same block.

**Definition**

The **block miss ratio** is the ratio of the number of blocks accessed from main memory to the total number of blocks accessed.
Lemma 4.2 (Hayes's formula)

For a given block miss ratio $M_b$ and a packing factor $N_p$; the cache miss ratio $S$ is:

$$S = \frac{M_b}{N_p}$$

Proof

Let $B_1$ and $B_2$ be the number of blocks accessed in the cache and main memory respectively. The block miss ratio is then:

$$M_b = \frac{B_2}{B_1 + B_2}$$

By definition, the cache miss ratio is the ratio of the number of memory references to main memory, $N_m$, to the total number of memory references, $M_r$. Since an access to a main memory block is a result of a cache miss, after which the whole block is moved to the cache where the rest of the accesses to that block are made. Thus $N_m$ is equal to $B_2$, and $S$ becomes:

$$S = \frac{N_m}{M_r} = \frac{B_2}{M_r}$$

By definition $N_p = M_r / (B_1 + B_2)$; thus $M_r = N_p \times (B_1 + B_2)$. Therefore:

$$S = \frac{B_2}{N_p \times (B_1 + B_2)} = \frac{M_b}{N_p}$$

[Q.E.D]
The previous lemma insinuates that decreasing the block miss ratio, or increasing the packing factor results in a lower miss ratio. In the preceding chapter we demonstrated the ability of the management models in reducing the block miss ratio through preloading and variable classification. On the other hand, the packing factor could be increased by applying one of the several packing (clustering) methods used for the packing of pages in main memory.

In the following section we shall make a thorough analysis of the complexity of optimal packing algorithms. Our study will be divided into three major parts. We shall first look at block packing methods for a one block cache. We shall then look at the number of possible ways to pack for the one block cache followed by the study of the complexity for one optimal packing algorithm. Our aim then is to extend these results to caches with two or more blocks.

In the third section of this chapter, we shall present the prerequisites for the implementation of the various loop transformations discussed in the previous chapter. Finally, we shall do some profitability analysis of our models under various execution conditions.

4.2. Packing For The One-Block Cache

For the analysis of optimal performance we shall assume the simple model of a cache with one block. For this model $M_b$, the block miss ratio, is one since every element in the block reference string causes a fault. Consequently, the cache miss ratio becomes:
\[ S = \frac{1}{N_p} \]

This equation makes \( S \), the miss ratio, depend only on the packing factor \( N_p \). Maximizing the value of \( N_p \) would effectively optimize the system's performance of the simple model.

From lemma 4.2 we learned that the value of the packing factor is equal to the ratio of the number of memory references to the number of block transition reference string. The number of memory references is a program invariant, thus maximizing \( N_p \) requires the minimization of the number of block transitions. Furthermore, for a given packing the number of block transitions is fixed and is independent of the cache size.

In the one-block cache each miss forces the replacement of the whole cache. Under such a scheme the cost of a memory reference string might be defined as the number of cache misses caused by that string. In the one-block cache model, a miss occurs whenever there is a transition from a reference in one block to a reference in another block. For example, for the memory reference string:

\[ ABCDABABCABACDC \]

If the block size is two then we can pack \( AB \) into one block and \( CD \) into another then the cache misses would occur at the places marked:

\[ |AB|CD|AB|C|ABA|CD \]

Consequently the optimal packing for the one block cache is that which minimizes the number of transitions. One way of tackling the
minimization problem is by transforming it into a graph problem.

For a given memory reference string we can build a graph that would reflect a particular relation in the string, like the neighborhood property. Here each vertex in the graph represents a variable and each edge between the two vertices has a weight equal to the number of times the two variables appear together in the reference string. The resulting graph is called a proximity graph. The proximity graph of the previous reference string is:

```
     C
  /     \
 /       \
A - 6 - D
     /     \
    1     2
     /       \
     B - 3   
```

Graph 1

For this graph the best packing is the one which cuts edges of minimum total weight. In other words, we subdivide the the graph into subgraphs, each with the same number of vertices such that the aggregate edge weight for the subgraphs is maximal. The dual of this problem, called the G-partition, has been shown to be NP-complete, and we shall later show that our packing problem is reducible to it. But first we shall look at simple approaches for tackling proximity graph problems.

In the preceding chapter, I discussed a similar packing (clustering) method as perceived by Hatfield and Gerald [71]. An analogous clustering techniques have also been proposed by Ferrari [74].
Both of these methods employ a clustering process called hierarchal classification.

Hierarchal classification requires a matrix whose entries reflect the value for a particular criteria between the data element. This matrix is symmetrical with zeroes on the diagonal, and it is sometimes called a proximity matrix.

Dubes and Jain ['80] discussed many algorithms for hierarchal clustering. Among these algorithms are: the Single-link [Sibson '73], the complete-link methods [Defays '77], and the group average method [Ward '63]. These algorithms try to minimize the dissimilarity values within each cluster.

For the packing problem we need to maximize the similarity values within each cluster. For this reason we have to do some minor changes within the clustering algorithms. This could be accomplished by reversing the condition for the fusing of two elements in the proximity matrix.

**Example:**

For the following reference string R:

```
1 2 3 2 1 5 2 5 3 2 6 4 6 5 3 2 5 2
```

we have a proximity matrix with the neighborhood relation:

```
2 3 4 5 6
1 2 0 0 1 0
2 3 1 4 1
3 0 2 0
4 0 2
5 1
```
We shall use a modified Single-link clustering method for packing. This is a Greedy type algorithm which at every step searches for the two elements having maximum value and group them together.

\[(2,5) \ 3 \ 4 \ 6\]

\[
\begin{array}{cccc}
1 & 2 & 0 & 0 \\
(2,5) & 3 & 1 & 1 \\
3 & 0 & 0 \\
4 & 2 \\
\end{array}
\]

\[(2,3,5) \ 4 \ 6\]

\[
\begin{array}{cccc}
1 & 2 & 0 & 0 \\
(2,3,5) & 1 & 1 \\
4 & 2 \\
\end{array}
\]

\[(4,6)\]

\[
\begin{array}{cccc}
(1,2,3,5) & 1 & 1 \\
4 & 2 \\
\end{array}
\]

\[(1,2,3,5,4,6)\]

To use these algorithms for the packing problem, we simply remove any element from the proximity matrix when it reaches the required cache size (one-block cache).

For the packing of a block size of three we have \{(2,3,5),(1,4,6)\} and for a block size of four we have \{(1,2,3,5), (4,6,\#,\#)\} where '\#' and '*' stand for any variable. []
Both of these packings are optimal, but this may not be true for larger or different reference string. With the proximity matrix, all clustering algorithms make the same number of steps. First they make \( n \) decisions (or comparisons) then the number of decisions reduces by one after each step. Therefore: \( n(n+1)/2 \) is the total number of decisions made by any of the algorithms using a proximity matrix, or simply of \( O(n^2) \). To find the optimal value, all possible paths must be investigated. The number of decisions in this case would be \( (n!) \), and the algorithm is \( O(n!) \).

4.3. The Complexity Of Packing

Kernighan's packing algorithm produces the minimal block transition, but it is only applicable for the packing of program code [Kernighan '71]. In our case we are mainly interested in packing data elements which do not meet Keringhan's conditions. Baer, showed that not abiding to one of Kerighan's condition could result in a better packing [Baer '76]. In this section I shall discuss the complexity of the optimal packing algorithm, that which minimizes the number of the block transitions, for a program with \( M \) distinct variables.

First, we need to find the number of ways we could pack \( M \) distinct variables for a one block cache of size \( k \). To formalize the question, I shall rephrase it to a set theory problem:

For a given set of size \( M \) what is the number of possible partitions such that the subsets within each partition are of size \( k \)? How do we generate these partitions?
For solving this problem, I constructed all possible subsets that could be members in any of the partitions. Using these subsets as elements, I then built the required partitions.

Example

For the set $S=\{a,b,c,d\}$ there are six subsets of size two:

\[
\{a,b\} \{a,c\} \{a,d\} \{b,c\} \{b,d\} \{c,d\}
\]

And there are three possible partitions:

\[
\{\{a,b\},\{c,d\}\} \{\{a,c\},\{b,d\}\} \{\{a,d\},\{b,c\}\}
\]

For a one block cache of size two we have only three ways for packing four distinct variables. The one which maximizes $N_p$ is selected for implementation.

Definition:

For a given set the fixed element subsets (FES) are the subsets which are equal in size and whose intersection is a unique element of the original set.

Example:

For the set $S=\{a,b,c,d\}$ the FES for $\{a\}$ are:

\[
\{\{a,b\},\{a,c\},\{a,d\}\}
\]

Lemma 4.3

Each element in a set has the same number of FES.
Proof:

For subsets of size $K$, and a set $S$ of size $M$, the FES of an element $A_i$ is constructed as follows:

Let the set $T=S-A_i$. Form all combinations of $T$ taking $K-1$ of its elements at a time. Therefore there will be $\binom{M-1}{K-1}$ distinct subsets of $T$ of size $K-1$.

Let $V = \binom{M-1}{K-1}$.

Let the subsets of $T$ of size $K-1$ be called $B_j$ for $0<j<V$. The FES of $A_i$ are: $\{A_i\} \cup \{B_j\}$ for $0<j<V$. Therefore, the number of subsets in an FES is $V$. Taking another element of $T$ will also yield an FES with $V$ elements [by construction]. Thus all FES for a given set are equal in size. [Q.E.D]

In what follows I will assume that $M$ is a multiple of $K$.

Theorem 4.1

For a set of size $M$ the number of partitions $P$ for this set such that the subsets within a partition are all of size $K$ and $M = NK$ is:

\[ P = \prod_{n=0}^{N-1} \binom{M-nK-1}{K-1} \]

Proof

I prove this theorem by induction on $M$. Since $M$ takes only multiple values of $K$, assume $M = K$ (equivalent for the case of 1) then for $M=K$ we have:

\[ P = \binom{K-1}{K-1} = 1 \]
If the size of the subset within a partition is equal to the size of the set then the partition must have only one subset which is the set itself and therefore all the partitions for this set are equal and we have only one distinct partition namely the set itself; therefore (1) is true for \( M=K \).

Now assume the equation is true for some value \( M \). We will show it is true for \( M+K \) (equivalently if it is true for \( n \) we will show it is true for \( n+1 \)).

For a set \( S \) of size \( M+K \) we pick an element \( A_1 \) we then construct the FES for it. Every fixed element partition of \( S \) will include one subset that will have the element \( A_1 \) (property of the partition). This subset will be a member of the FES of \( \{A_1\} \) (by the construction of the FES and Lemma 4.3). Let the members of the FES be called \( B_j \) for \( 0 < j < V \) where \( V \) is given by:

\[
V = \binom{M+K-1}{K-1}
\]

The number of fixed element subsets of size \( K \).

To construct the partitions for \( S \) we simply take the union of each \( B_j \) with all the partitions of \( \{S\} - \{B_j\} \). I claim that this will produce all the partitions for \( S \). Assume there exist a partition call it \( p \), not equal to any of the partitions constructed. The partition \( p \) must have a subset , call it \( f \), containing the element \( A_1 \) of \( S \) (property of partition), this subset must be a member of the FES for \( A_1 \) (property of the construction of FES). Let \( f = B_g \) then the remaining \( p-f \) subsets must be in one of the partitions of \( S-B_g \) (by assumption
that equation (1) is true for a set of size \( M \) and the property of the partitions being distinct). Therefore the partition \( p \) is in the set of partitions \( B_j \cup \{ \text{partitions of } [S-B_j] \} \) for \( 0 < j < V \). Hence this construction gives all the partitions of the set with \( M+K \) element. The number of partition is then equal to \( V \) times the number of partition for \( M \).

\[
P = \binom{M+K-1}{K-1} \prod_{n=0}^{K} \binom{M-nK-1}{K-1}
\]

Therefore

\[
P = \frac{(M+K-K)}{\prod_{n=0}^{K} \binom{(M+K)-nK-1}{K-1}}
\]

[QED]

Thrall ['81] proposed another formulation for the partition problem. Here the first subset in the partition is formed by taking all possible combinations of \( M \) elements taken \( K \) at a time. Similarly the second subset in the partition is formed by taking combination of \( M-K \) elements taken \( K \) at a time. However, we shall get all possible permutations for the required partition. Thus we divide by the number of possible permutations which is \( N! \), where \( N = M/K \). In the following corollary I shall prove the equivalence of both formulation.
Corollary 4.1.1

For $M$ a multiple of $K$: $M = NK$

\[
\frac{\prod_{n=0}^{M-K} (M-nK-1)}{\prod_{n=0}^{K} (M-nK)} = \frac{\prod_{n=0}^{M-K} (M-nK)}{N!}
\]

Proof

Since $M$ is a multiple of $K$ then:

$N! = N(N-1)(N-2)\ldots(1) = (M/K)((M-K)/K)\ldots(K/K)$

The left hand side is equal to:

\[
\begin{pmatrix} M - 1 \\ K - 1 \end{pmatrix} \begin{pmatrix} (M-K) - 1 \\ K - 1 \end{pmatrix} \ldots \begin{pmatrix} K - 1 \\ K - 1 \end{pmatrix}
\]

which is equal to

\[
(M/K) \ast \begin{pmatrix} M - 1 \\ K - 1 \end{pmatrix} \ast ((M-K)/K) \ast \begin{pmatrix} (M-K) - 1 \\ K - 1 \end{pmatrix} \ast \ldots \ast (K/K) \ast \begin{pmatrix} K - 1 \\ K - 1 \end{pmatrix}
\]

--------------------------------------------------------------------------------------------------

$N!$

But $(M/K) \ast \begin{pmatrix} M - 1 \\ K - 1 \end{pmatrix} = \begin{pmatrix} M \\ K \end{pmatrix}$. Therefore
\[
\frac{M-K}{K} \prod_{n=0}^{N-1} \binom{M-nK-1}{K-1} = \frac{M-K}{K} \prod_{n=0}^{N-1} \binom{M-nK}{K} \frac{N!}{N!}
\]

[Q.E.D.]

**Corollary 4.1.2**

For \( M \) a multiple of \( K \) ( \( M=NK \) )

\[
\prod_{n=0}^{N-1} \binom{M-nK-1}{K-1} \text{ is of order } O\left( N^{(M-n)} \right)
\]

**Proof**

Using the second formula given in corollary 4.1.1 we get:

\[
\begin{align*}
\binom{M}{K} \binom{M-K}{K} \cdots \binom{K}{K} &= \frac{M!}{K!} \frac{(M-K)!}{K!} \cdots \frac{K!}{K!} \\
&= \frac{M!}{K!} \frac{(M-K)!}{K!} \cdots \frac{K!}{K!} \left( \frac{1}{K+1} \right)^N \left( \frac{1}{K+2} \right)^N \cdots \left( \frac{1}{K+N} \right)^N \\
&= \frac{N!}{N!}
\end{align*}
\]

Which is equal to \( \frac{M!}{(K!)^N N!} \).

Using Stirling's approximation [Knuth '69] for \( n! \):

\[
n! \approx \sqrt{2\pi n}(n/e)^n
\]
Thus the formula becomes:

\[ \sqrt[k]{(e/\sqrt{2\pi k})^N} \times N^{N(k-1)} \]

[Q.E.D]

**Example**

<table>
<thead>
<tr>
<th>K</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>---</td>
<td>1</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>10</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>12</td>
<td>10395</td>
<td>15400</td>
<td>5775</td>
<td>---</td>
</tr>
</tbody>
</table>
| 20 | 6.5x10^8 | --- | 2.5x10^9 | 4.88x10^8 | []

This example reflects the complexity of the packing problem. For a given reference string we need to find simpler methods for achieving the optimal packing without generating all partitions and testing each one of them to see if it is the optimal.

An alternate approach is to construct the subsets that give high packing factor then construct a partition from them. With this plan in mind we look at all the neighbors of each variable in the given reference string. Subsets are then constructed by choosing variables that tend to be neighbors more often and place them in the same subset. From these subsets we then build the partition that might yield optimal packing. This would reduce the size of the problem but it will still be hard.
Any proximity matrix could be represented by an undirected weighted graph $G \{V, E\}$, with the set of variables represented by the set of nodes $V$. The weight on each edge reflects the number of times the variables (nodes) pertained a particular relation. For example, the weight on the edge could reflect the number of times the variables were neighbors in the memory reference string. Similarly any weighted undirected graph could be represented by a proximity matrix.

**Definition**

Let $G$ be undirected graph having vertex set $V$ and edge set $E$ with a value for each edge. A k-partition [Brucker '78] is the partitioning of $V$ into $k$ nonempty disjoint sets. If a k-partition results in $k$ isomorphic subgraphs then we have a G-partition [Kirkpatrick and Hell '78]. The weight of a partition is the sum of all the weight of the edges of the subgraphs:

$$W_{\text{total}} = \sum_{m=1}^{d} \sum_{e \in E(G_m)} c(e)$$

**Theorem 4.2**

Finding the G-partition of minimum weight is an NP-complete problem.

**Proof**

[Kilpatrick and Hell '78]
Theorem 4.2

Finding the optimal packing using a proximity matrix is an NP-complete problem.

Proof

Our strategy is to first show that the given problem (call it L2) is NP-hard, then exhibit a polynomial time nondeterministic algorithm for it.

The G-partition problem (call it L1) is NP-hard [Kilpatrick and Hell '78].

For a given weighted undirected graph (any instance of L1), we negate the weights and then build a proximity matrix. With this proximity matrix we find the optimal packing: that which maximizes the total weight of all the partition subsets. Since each of the partition subsets has equal number of (nodes), and if edges with zero weight are permitted then the partition of the packing problem is a partition of the given graph (of L1) with the subgraphs being isomorphic (all the subgraphs are complete graphs and having the same number of nodes). Negating the value obtained from the packing results in the minimal value partition or simply the minimal G-partition. All these transformations could be done in polynomial time of order $O(M + E)$ the number of variables and edges respectively.

From the previous argument L1 is polynomial time reducible to L2. Therefore L2 is also L2 is NP-hard.
Suppose there is a conjectured optimal packing value \( W \) for an instant of a packing problem requiring the packing of \( M \) variables to blocks of size \( K \). Then there exists a nondeterministic algorithm which guesses a packing \( P \). The checking algorithm \( A \) will simply have to compute all possible pair combination within each packed block, which is \( K(K-1)/2 \). Since there are \( N \) of these blocks, the algorithm has to check for \( N*K*(K-1)/2 \) values from the proximity matrix and then perform the summation and comparing it to \( W \) to give an answer of Yes or No. Because \( M=N*K \) and \( K \) is generally assumed to be constant, the verifying algorithm \( A \) is thus of \( O(M) \) which is polynomial in \( M \). [Q.E.D]

4.4. The N Block Cache

Our aim in this analysis is to find the packing that results in the minimal number of cache misses. The one block cache simplifies this analysis by requiring that we find the optimal packing. If the optimal packing for the one block cache (least number of block transitions), also results in the minimal number of cache misses for a two block cache assuming that we apply an optimal replacement policy, then the complexity of finding the minimal number of cache misses depends on the complexity of the one block packing. This result could then be extented to \( N \) block cache.

The next question of interest is whether the optimal one block packing will always yield the best performance for a cache system with two or more blocks using the same or different replacement polices. The answer of this question is illustrated in the next lemma.
Lemma 4.4

For a given memory reference and a cache with two or more blocks, there may exist a suboptimal one block packing which may contribute less cache misses than the optimal one-block packing.

Proof

The proof is given in the next example.

Example

121218178787834567878781812121765

For the given memory reference string, the optimal one block packing is:

A={1,2,5} B={8,7,3} C={4,5,6}

The block transition is represented by the following string:

A B A B C B A B A B C

Its length is 11 or equivalently there were 11 cache misses for a cache system with one-block. Here no replacement policy is needed since at every cache miss the block in the cache is replaced.

We shall test this packing in cache system with two blocks, first using Belady's optimal replacement policy and then the LRU.
For a two block cache and using the optimal replacement policy we get:

* A A C A C
* * B B B B

Five cache misses (those underlined).

Using the same cache but with the LRU:

* A B A B C B A B A B C
* * A B A B C B A B B B

This also result in five cache misses.

For a suboptimal packing:

\[ A = \{1, 2, 8\} \quad B = \{\#, 3, 4\} \quad C = \{5, 6, 7\} \]

The block transition string for the one-block cache is:

A C A C A C A C A C A

Which is of length 15 i.e. 15 cache misses for the one block cache.

However, for a two block cache and using the optimal replacement we only get 4 cache misses!

* A A B A
* * C C C
On the other hand in using the LRU we get 5 cache misses!

* A C A C A C A B C A C A C A
* * A C A C A C A B C A C A C A

[]

The previous example suggest that the optimal packing for the one-block cache, when used on a cache with two or more blocks, will not guarantee the lowest number of cache misses. This makes the problem minimizing the number of cache misses by way of packing even more complex.

In the following example we shall illustrate the usefulness of our models in reducing the block traffic in the cache main memory bus.
Example

Assume we have a cache with three blocks. Let the block size be 3 words. Given the following sequence of statements:

\[ E := A \times B + 2 \times D \]
\[ F := A \times B + 1 \]
\[ G := 4 \times D \]

The quadruples are:

\[ t_1 := A \times B \]
\[ t_2 := 2 \times D \]
\[ t_3 := t_2 + t_1 \]
\[ E := t_3 \]
\[ t_4 := A \times B \]
\[ t_5 := t_4 + 1 \]
\[ F := t_5 \]
\[ t_6 := 4 \times D \]
\[ G := t_6 \]

The reference trace generated from this code is:

BA(t1)D(t2)(t1)(t2)(t3)(t3)EBA(t4)(t4)(t5)(t5)FD(t6)(t6)G

An optimal packing is:

\[ 1 = \{ A, B, E \} \quad 2 = \{ t_1, t_2, t_3 \} \quad 3 = \{ t_4, t_5, F \} \quad 4 = \{ D, t_6, G \} \]

The block reference string is:

1 2 4 2 1 3 4

Using optimal replacement we get four cache misses.

* 1 1 1 1
* * 2 2 4
* * * 3 3

However we have four modified blocks that all write-in cache systems will have to copy back; thus adding to the CPU idle time.
Most of the cache misses are usually caused by array elements. In the foregoing section we shall present conditions for doing various loop transformations which have been illustrated to be useful in reducing the number of cache misses.

4.5. Conditions For Loop Transformations

Loop restructuring for reducing a program's memory requirements [Abu-Sufah '78], or the vectorization of statements within a loop [Kennedy '81] are always preceded by a dependency analysis.

Two statements in a program are either not related or have one of the three types of dependencies between them: true dependence where the output of the first statement is used in the second statement, antidependence where a variable used in the first statement is redefined in the second statement, or output dependence where the variable is defined in both statements.

Intuitively, the order of the statements determines the type of dependency. If the statement depended upon precedes the depending statement, then we have normal dependence. In a straight line code is the typical form of dependence, but in a loop a may depend on a statement which follows it in the program text. Moreover, such a dependency may have only risen due to the statements being in the loop. If the loop is removed and the index variable is given any constant value then such dependency would disappear. I call this type of dependence a carried dependence.
Such classification is helpful in the understanding of many dependency concepts discussed in Kennedy's paper, and will help in the formalization of the conditions for loop distribution that have been presented in chapter two.

Definition

Given the two statements S1 and S2:

\[ S1 \quad X[f(i)] = T(...) \]
\[ S2 \quad a = G(X[h(i)]) \]

Normal dependency:

Normal true dependence between S1 and S2 occurs if:

1. S1 comes before S2 \((S1 < S2)\),
2. The variable (array element) defined in S1 (modified) is used in S2 (accessed),
3. In a loop the following must hold:
   \[ f(i_1) = h(i_2) \text{ for } 1 \leq i_1 \leq i_2 \leq N. \]

Normal antidependence between S1 and S2 occurs if:

1. S1 comes after S2 \((S1 > S2)\),
2. The variable (array element) used in S2 is redefined (modified) in S1,
3. In a loop the following must hold:
   \[ f(i_2) = h(i_1) \text{ for } 1 \leq i_1 \leq i_2 \leq N. \]
Carried dependency

Carried true dependence between S1 and S2 occurs if:

(1). S1 comes after S2 (S1 < S2),

(2). The variable (array element) defined in S1 is used in S2.

(3). In a loop the following must hold:

\[ f(i_1) = h(i_2) \text{ for } l \leq i_1 < i_2 \leq N. \]

Carried antidependence between S1 and S2 occurs if:

(1). S1 comes before S2 (S1 < S2),

(2). The variable (array element) used in S2 is redefined in S1.

(3). In a loop the following must hold:

\[ f(i_2) = h(i_1) \text{ for } l \leq i_1 < i_2 \leq N \]
Definition

Given the two statements $S_1$ and $S_2$ of the form:

$S_1 \quad X[f(i)] = T(...)$
$S_2 \quad X[h(i)] = G(...)$

Normal dependency:

Normal output dependency between $S_1$ and $S_2$ occurs if:

1. $S_1$ comes before $S_2$ ($S_1 < S_2$),
2. The variable (array element) defined in $S_1$ is redefined in $S_2$,
3. In a loop the following must hold:

   $f(i_1) = h(i_2)$ for $1 \leq i_1 \leq i_2 \leq N$

Carried output dependency between $S_1$ and $S_2$ occurs if:

1. $S_1$ comes after $S_2$ ($S_1 > S_2$),
2. The variable (array element) defined in $S_1$ is redefined in $S_2$,
3. In a loop the following must hold:

   $f(i_1) = h(i_2)$ for $1 \leq i_1 \leq i_2 \leq N$

[]
Example

Normal Dependence  Carried Dependence

-----------------------------------------------

True dependence

\[
\begin{align*}
&D0 \ 10 \ i = 1 \ to \ 100 & & D0 \ 10 \ i = 1 \ to \ 100 \\
&S1 \quad X[i+2] = T(...) & & S2 \quad a = X[i] \\
&S2 \quad a = X[i] & & S1 \quad X[i+1] = Y(...) \\
10 & \text{CONTINUE} & & 10 \quad \text{CONTINUE}
\end{align*}
\]

-----------------------------------------------

Antidependence

\[
\begin{align*}
&D0 \ 10 \ i = 1 \ to \ \infty & & D0 \ 10 \ i = 1 \ to \ 100 \\
&S2 \quad a = X[i+2] & & S1 \quad X[i] = Y(...) \\
&S1 \quad X[i] = T(...) & & S2 \quad a = X[i+1] \\
10 & \text{CONTINUE} & & 10 \quad \text{CONTINUE}
\end{align*}
\]

-----------------------------------------------

Output dependence

\[
\begin{align*}
&D0 \ 10 \ i = 1 \ to \ 100 & & D0 \ 10 \ i = 1 \ to \ 100 \\
&S1 \quad X[i] = T(...) & & S2 \quad X[i] = Y(...) \\
&S2 \quad X[i] = Y(...) & & S1 \quad X[i+1] = T(...) \\
10 & \text{CONTINUE} & & 10 \quad \text{CONTINUE}
\end{align*}
\]

-----------------------------------------------
Dependencies between statements in a program are represented by a directed graph. The set of statements are represented by the vertices, and the set of directed edges represents the dependency relation. The edges are labeled with the type of the dependency. At the tail of the edge is the statement where the dependency initiated i.e. the first, of the two statements, to be executed. If we assume our programs have of a sequence of statements none of which is a go to statement, then the of flow control travels from one statement to the next in the order given. A subsequence of statements, containing one or more statements, is executed either one time and is labeled as straight line code, or possibly more than one time and the subsequence is labeled as a loop.

Lemma 4.5

Two statements cannot have a carried dependence between them unless both are within the same loop structure.

Proof

From the definition both the normal dependency relation, and the flow of control have the same direction. Thus normal dependency could exists both in straight line code, and loops. On the other hand, the direction of the carried dependency relation is opposite to that of the flow of control. Thus, unless the sequence of statements where the dependency might appear is executed more than once (till the dependency is established i.e. \( f(i_1) = h(i_2) \)) the dependency relation is non-existent there. [QED]
Definition

A recurrence relation between two statements occur if there exist a normal and a carried dependencies between the two statement with each of the statements being the origin of one of the dependencies.

Lemma 4.6

A recurrence could only occur within loops

Proof

Using lemma 4.5 carried dependencies can only be within a loop, and by definition a recurrence must include a carried dependencies.

[QED]

Originally we thought that existence of a recurrence prevents us from doing any of the loop transformations. In the following theorem we shall show that because of the carried dependency a loop transformation could not be performed. As a consequence, a recurrence breaking transformation tend to eliminate the carried dependency part of the recurrence.

Definition

The index variable of the loop that included a carried dependency is called a carrier index.
Definition

A horizontal distribution is safe if all dependencies are preserved.

Theorem 4.4

A horizontal distribution applied to two statements within a loop is safe if and only if there is no carried dependency between them.

Proof

Case 1

Here we shall prove that if the horizontal distribution is safe then all dependencies are preserved. In particular, if there was a loop carried dependency then after the distribution there must be this dependency. However, after the distribution the two statements are in different loops and by lemma 4.5 cannot have a loop carried dependency between them thus contradicting our assumption of safety. Hence, there could not be a carried dependency between the two statements.
**Case 2**

Assume that there are no loop carried dependencies, then we have to show that normal dependencies and loop carried dependencies are preserved. First, if the loop is horizontally distributed then the two statements will in different loops and thus by definition have no loop carried dependencies between them, therefore this class of dependencies is preserved.

In the original loop if there was no carried dependency then there is either normal dependency or no dependency between the two statements. First we assume there is normal dependency, we shall prove that it will be preserved.

Since, by assumption, there was normal dependency before the horizontal distribution then the three properties given in the definition of normal dependency must have been true. We shall prove that are still true.

Horizontal distribution does not change the order of the statements of the original loop, nor does it change the functions $f$ and $h$. Furthermore, the limits in the distributed loop will the same as that of the original loop. Hence, properties one and three of the definition are preserved by the horizontal distribution.

For normal true dependency, if in the original loop there were one or more definitions of a variable in $S_1$ all of which have preceded the use of the variable in $S_2$, then the last definition
of the variable will be used by S2. This will also be true after the distribution. However, if the variable defined in S1 and later used in S2, is redefined again in S1, then in the original loop the first definition of the variable would reach S2, while in the distributed loop the second definition (last) by S1 would reach S2. This makes distribution unsafe. On the other hand, since there is a use of the variable in S2 followed by a definition of that variable in S1 i.e. \( f(i_2) = h(i_1) \), \( 1 \leq i_1 < i_2 \leq N \), and since S1 precedes S2 then there is a carried antidependency in the original loop. But that contradicts our assumption of no carried dependencies between the two statements. Therefore, all definitions of the variable must precede the use of that variable in S2. Hence horizontal distribution preserves normal true dependence in the absence of carried dependencies.

Similarly with normal antidependence, all uses of the variable in S2 must precede any definitions of that variable. If there was a use of the variable in S2 followed by a definition of the variable in S1 followed by another use of the variable in S2, then this definition and the second use in S2 \( (i_1 < i_2) \) means that the original loop has a carried true dependence which contradicts our assumption. Therefore, all the uses for the variable in S2 precedes all the definitions in S1. Under such conditions the normal antidependency is preserved.
For normal output dependency the variable is first defined in S1, and then redefined in S2. Thus in the original loop the last definition by S1 would reach S2 and the variable is redefined. Horizontal distribution would preserve this order. However, if there was a definition in S1 followed by a definition in S2 and later another definition by S1, then in the original loop the first definition reaches S2. But after the distribution the second (last) definition reaches S2. However, this means that the variable defined in S2 is redefined in S1. Thus the original loop has a carried output dependency contrary to the fact. Thus such situation would not arise and the horizontal distribution will be safe.

Assume that although there was no normal dependency between the statements in the original loop, after the horizontal distribution there existed normal dependencies. Assume that there was normal true dependence after the distribution. This means \( f(i_{\text{sub 1}}) = h(i_{\text{sub 2}}) \). Since there was no normal true dependence \( i_1 \) is not less than \( i_2 \) and therefore \( i_1 \) is greater than \( i_2 \). Under such conditions the original loop has a carried antidependence which contradicts our assumption. Hence, if there were no normal dependencies in the original loop there will not be any normal true dependencies after the distribution. The proof for the other cases follow the same logic. Therefore, with no carried dependencies horizontal distribution could be done safely. [QED]
Corollary 4.4.1

Any loop that could be horizontally distributed could also be vertically distributed.

Proof

We shall prove this theorem by construction. First, we strip mine the loop with the limits in the inner loop equal to the one used for the vertical distribution. Since strip mining changes the indexing scheme which by itself does not change the order of execution nor the actual order of the statements nor the range of indices, it should not then change any of the dependencies.

It is given that the loop could safely be horizontally distributed, and by theorem 4.4 there is no carried dependencies between the statements in the loop. Suppose for one set of limits of the inner loop we get a true carried dependency (a similar test holds for carried antidependency). Then \( f(i_2) = h(i_1) \) for \( ILB \leq i_1 < i_2 \leq IUB \). But since \( 1 \leq ILB < IUB \leq N \), then \( 1 \leq i_1 < i_2 \leq N \). This means that we had a carried dependency in the original loop which is contradiction. Therefore, for all the iterations of the outer loop, of the vertical distribution, there will never be an inner loop whose statements have a carried dependency. On that account we can make a safe horizontal distribution of the inner loop. By this procedure we have vertically distributed the loop. [Q.E.D]
In the following lemmas we show how the compiler detects the existence of a carried dependency.

**Lemma 4.7**

A normal dependency between two statements due to scalars is a recurrence if the two statements are within a loop.

**Proof**

In both statements the scalar causing the dependency uses the same memory location. Thus if there is a normal true dependency, then in the loop there is also a carried antidependency between the two statements (since it satisfies the definition of carried antidependency), and therefore we have a recurrence. Similarly for other types of normal dependencies. [Q.E.D]

**Lemma 4.8**

The two statements S1 and S2 have a carried true dependence between them if S2 is before S1, and S2 uses the variable defined in S1, and the following two tests are satisfied:

If \( f(i) = a_0 + a_i \) and \( h(i) = b_0 + b_i \) then,

a) \( \{ \gcd (a_1 - b_1) \} | (b_0 - a_0) \)

b) \( -b_1 - (a_1 - b_1)^+ (N-2) < b_0 + b_1 - a_0 - a_1 < -b_1 + (a_1 - b_1)^+ (N-2) \)
Proof

{Theorem 4 in [Kennedy '81].}

Definition

The fusing of two loops, having the same limits, is said to be safe if the new loop preserves previous dependencies and does not introduce any new dependencies.

Theorem 4.5

The fusion of two loops, each containing a single statement, is safe if and only if the fusion does not introduce any carried dependency.

Proof

Case 1

Since the two statements are in different loops then by lemma 4.5 they have no carried dependency between them. Assuming that the fusion of the two loops is safe then by definition no new dependencies are introduced and therefore no carried dependency.

Case 2

We will now show that if the fusion does not introduce any carried dependency then it is safe.
Since by lemma 4.5 there was no carried dependency between the statements because of being in different loops, then the carried dependencies are preserved and no new carried dependencies are introduced. Now we check for the preservation of normal dependency, and that no new dependencies are introduced.

Now suppose after fusion there existed another normal dependency that did not exist before. On the other hand by assumption we do not have any carried dependency, thus by theorem 4.4 we can have a safe horizontal distribution for the fused loop and normal dependency must also be preserved including those introduced by the fusion process. The order of the statements did not change, nor the functions f and h and nor the ranges. This means that this dependency if existed in the fused loop must have also been in the original unfused loops. [Q.E.D]

**Corollary 4.5.1**

Two loops that could safely be fused together could also be vertically distributed.

**Proof**

First we fuse the two loops. By theorem 4.5 the statements within the fused loop do not have a carried dependency between them. Therefore, by theorem 4.4 we could have a safe horizontal distribution for the fused loop. On that account, and using theorem 4.5 we could vertically distribute the fused loop. [Q.E.D]
Definition

The **distance** of a dependency is:
\[ d(i) = | f(i) - h(i) |. \]

Definition

The **stretch** of a vertical distribution is difference between upper and lower bounds of the inner loop plus one:
\[ |\text{IUB} - \text{ILB} + 1 | \]

**Theorem 4.6**

If the minimum distance of a carried dependency is a constant \( K \), \( \min D(i) = K \), then we can vertically distribute the loop with a stretch \( j < K \).

**Proof**

First we strip mine the loop. The inner loop could safely be horizontally distributed if for all the iteration of the outer loop, of the vertical distribution, there will never be an inner loop whose statements have a carried dependency. Suppose for one set of limits of the inner loop we get a carried true dependency (the test are similar for a carried antidependency). Then \( f(i_1) = h(i_2) \) for \( \text{ILB} \leq i_1 < i_2 \leq \text{IUB} \). However, \( f(i_1) - h(i_2) = K - (i_2 - i_1) \)

Since \( (i_2 - i_1) < (\text{IUB} - \text{ILB}) < K \), then \( f(i_1) - h(i_2) > 0 \). Which contradicts our assumption. Therefore the third condition of carried dependency is never satisfied and hence there will never be a
carried dependency for the inner loop with a stretch less than K. In consequence, we can make a safe horizontal distribution for the inner loop, and arrive at our intended vertical distribution. [Q.E.D]

Example

DO 10  I= 1 TO 100
    A(I) = C(I)
    B(I) = A(I + 10)
10  CONTINUE

The previous loop has a carried antidependency with a distance of 10. By theorem 4.6 we could vertically distribute this loop with a stretch of 10 or less.

DO 10  J1= 10 TO 90 STEP 10

    J2 = J1 + 10

    DO 20  I= J1 TO J2
        A(I) = C(I)
    20  CONTINUE

    DO 30  I= J1 TO J2
        B(I) = A(I + 10)
    30  CONTINUE

10  CONTINUE
A recurrence is broken if one of the essential dependencies is eliminated. If the aim here is to do any of the previous loop distributions then the carried dependency should be eliminated. Abu-
Sufah['79] and Kennedy ['80] give various ways for the breaking of carried dependencies.

4.5. Profitability analysis

In chapter one we described the cache as being cost effective. Moreover, the cache has been portrayed as being the economical solution to the speed mismatch problem between main memory and the CPU. In this section we shall formalize these concepts by analyzing the speed gain due to the cache. Thereafter, we shall analyze the speed gain in the implementation of the allocation models.

Kuck expressed the speed gain due to the cache as a function of the cycle times of main memory and the cache, and the hit ratio[Kuck '78]. Since all of our analysis are in terms of the miss ratio, we are inclined not use the hit ratio here.

\[ S_c = \frac{T_m}{T_e} = \frac{T_m}{(S \times T_m + (1-S) T_C)} \]

- \( S_c \) is the speed gain
- \( T_m \) is the main memory cycle time
- \( T_e \) is the effective cycle time
- \( T_C \) is the cache cycle time
- \( S \) is the miss ratio

Let \( r = \frac{T_m}{T_C} \). If we could write \( S \) as \( 1/(1+k) \) then \( S_c \) becomes:

\[ S_c = \frac{1+k}{1 + k/r} \]
Thus the further away main memory is the higher is the speed gain and the more cost effective is the cache.

The speed gain improvement \( S_i \) due to changes in the miss ratio is another useful parameter [Kuck '78].

\[
S_i = \frac{(S_c') - S_c}{S_c}
\]

For \( r \geq 1 \) if the miss ratio is reduced by 50 percent i.e. from \( 1/(1+k) \) to \( 1/2(1+k) \), then the speedup improvement becomes:

\[
S_i = \frac{1}{1+2(k+1)/(r-1)}.
\]

Thus a slight improvement in the miss ratio results in a noticeable improvement for systems with slower memories.

**Example**

Given two systems that are identical except for the speed of their main memory the first has \( r = 5 \) while the second has \( r = 41 \). If the miss ratio improvement for both was reduced from \( 1/20 \) to \( 1/40 \) (\( k = 19 \) then it is an improvement from \( 1/(1+k) \) to \( 1/2(1+k) \)).

For the system with \( r = 5 \) the value of \( S_i = 1/11 \) or simply a 9% speedup improvement.

For the system with \( r = 41 \) the value of \( S_i = 1/2 \) or simply a 50% speedup improvement.

To increase the systems performance the trend has been to increase the cache size. Recent development in memory technology has
reduced the price of main memory by almost two orders of magnitude, while the price of cache memory reduced only by one order of magnitude. This prompted system designers to make a choice between having a larger cache hence a faster response time or a larger main memory and hence a higher throughput.

Example

The Concept 32/87 is a new supermini computer [Mini-Micro '81], it has 1M byte for main memory ($T_m = 300$ ns), and a 16K byte cache ($T_c = 75$ ns). With the 16K cache the system achieves an effective cycle time of 153 ns, and 94.5 ns with a 32K cache. For the the 16K cache the speed gain: $S_c = 2$, but for the 32K cache $S_c = 3$. This makes $S_i = 50\%$ with the increase of cache size.

The increment of 16K byte for the cache costs $30,000$, while an increment of one million byte for main memory may cost $6,000$. Thus, for the price of increasing the cache to 32K we could increase main memory to 6M bytes.

In multi-programming environment larger memory results in lower page fault rate and consequently higher throughput, especially for large programs. On the other hand, a larger cache means a lower response time especially for on-line applications like image processing or programs that are CPU bound (require a lot of processing). Nevertheless, the relative increase in main memory is much higher than the relative increase in cache size, making the former more attractive in the general.
An alternative to increasing the cache size for achieving a better performance is needed here. We think that our cache allocation models make a reasonable alternative to the increase in cache size for achieving a better performance. Since our models are categorized as software they are more reliable than the hardware solution of augmenting cache size. Moreover, our models have negligible reproduction cost as compared to the extra cost in hardware.

As demonstrated in the previous chapter, the allocation models we effectively reduce the miss ratio for blocks containing scalar variables. In the following section we shall present an analysis of the performance gain by the allocation models during loop execution.

In the example presented in the previous chapter we demonstrated how the prompting model when used in conjunction with the vertical distribution could eliminate all the cache misses and the time for block write back. Our interest now is to investigate the gain or loss in CPU time as a result of this transformation.

If the number of the new loops created by the vertical distribution is L, then the number of extra indexing statements executed is equal to: \((L-1)N + 2(N/R)\), where \(N\) is the limit of the loop and \(R\) is the cache block size. The second element of the expression is due to the statements evaluating ILB and IUB.

Moreover, the number of Prompt and Release instructions is equal to \(A(N/R)\) and \(L'(N/R)\) respectively. If on the average these pseudo-instructions require the same time as an index instruction, then the total number of instructions executed \((I)\) is:
\[ I = N*(L-1) + 2*(N/R) + A*(N/R) + L'*(N/R) \]

or simply

\[ I = N*(L-1) + (N/R)(2+A+L') \]

If the number of arrays referenced in original loop is \( A \), then the total number of cache misses (load misses) of the original loop would be \( A*(N/R) \). The time to serve a miss under the write-in policy depends on the block chosen for replacement. If the block is not modified then the service time, cache miss penalty, would be equal to main memory access time. On the other hand, if the block to be replaced is modified then it has to be copied to main memory first before the request for the needed block is initiated; thus requiring twice the time of a read without a write back. For our loop there will be \( L'*(N/R) \) misses requiring a write back.

Let \( M \) be the number of instructions executed in a time taken to serve a cache miss (without a write back). Thus the total number of instructions that could have been executed if there were no cache misses is:

\[ J = (N/R)(A+L')*M \]

To have a performance gain with the management models, the value of \( I \) must be less than \( J \) or simply:

\[ M > \frac{(R*(L-1) + A + L' + 2)}{(L' + A)} \]

\[ M > 1 + \frac{(R*(L-1) + 2)}{(L' + A)} \]

Here we assumed for simplicity that \( N \) is equal to \( R*(N/R) \).
In the Amdahl 470 V6 at replacement modified is copied to a set of fast registers while requesting for the missing block from main memory. Thus the time for a write back is eliminated and the value of M should be:

\[ M > 1 + \frac{(R\times(L-1) + L' + 2)}{A} \]

This means that the further away main memory the better is the possibility for the success of the combined effort of restructuring and the management models. Notice that as R increases, the number of cache misses induced by the original loop decreases.

Example

The Amdahl 470 V6 has an average instruction time of 160ns and the cache miss penalty of about 640ns. If the index statements take one instruction to execute, then \( M = 4 \). For the example of the previous chapter:

\[ A = 6, L = 3, L' = 2. \]

If \( R = 8 \) then:

\[ 4 > \frac{1 + (8\times2 + 2 + 2)}{6} \text{ i.e. } 4 > 4.3 \]

which is not true. This means that with the prompting model the loop will take more time execute than without it!!

Nonetheless, technological trends favor a wider speed gap between CPU and main memory. A value for \( M > 10 \) or more has been envisioned with the implementation of the Josephson junction, making the implementation of our models an important feature to be added to
these high performance computers. []

In the analysis of the previous example we did not mention the advantages of our models in reducing contention, and the number of write back to main memory. The implementation of our models will be subject of the next chapter.

4.7 Conclusion

In this chapter we analyzed two enhancement schemes: optimal cache allocation through packing and loop transformations. From the analysis we concluded, to our dismay, that optimal cache allocation is a hard problem. Nonetheless, allocation is valuable tool. Therefore, we must seek heuristic techniques - the topic of the next chapter.
CHAPTER 5

Implementation Heuristics

5.1. Introduction

In chapter three of my thesis I presented the architectural features of the two cache management models. Details of the implementation of these two models were not fully pronounced. The examples presented were simple and did not depict actual characteristics of normal programs. However, the aim then was to exhibit the architectural attributes of the models rather than the aspects of their implementation.

For a given program, implementation of either model requires the analysis of data (variable) behavior within the program, followed by a restructuring of the program and a repacking of its data. The former is preceded by program flow analysis, while the latter requires dependency analysis.

The previous chapter covered dependency analysis and various loop restructuring methods. Here, I shall discourse on program flow analysis for the implementation of the two allocation models. A good survey of program flow analysis is given in [Kennedy '81]. A simple review of the main principles of flow analysis is presented in the appendix.

Before we impart any implementation details of the allocation models, we shall propose ways the compiler writer could implement for
improving cache performance, assuming the nonexistence of the hardware modifications for either model.

In chapter three we attributed a cost for each block movement between main memory and the cache. In our analysis we found that premature replacement, i.e. the replacement of blocks that might be used again, is the most costly. In the next lemma we show how premature replacements affect the cache miss ratio.

Lemma 5.1

If every program variable has a distinct memory address throughout the program execution, and when these variables are packed into blocks then at least one variable in each block is accessed, then:

\[ \# \text{ Cache misses} = \text{Total } \# \text{ of blocks} + \# \text{ Premature replacements} \]

Proof

Our first assumption that there is a distinct address for each of the program variables means that these variables will be packed into groups (cache blocks) and throughout the execution a variable will remain within the same group (cache block). Since in every program there is a fixed number of variables, the packing will result in a fixed number of cache blocks. Since each of these blocks is accessed at least once then we get the first argument of the summation. If the block is prematurely replaced then by definition this block will be requested later and would thus cause a cache miss. [Q.E.D]
If we assume that the addresses generated by the CPU are virtual memory addresses then each variable will have a distinct memory address. In general almost every variable is accessed during the execution phases of a program, and more definitely in every group of variables packed to the same block at least one of these variables gets accessed. Therefore, the previous lemma holds true for almost all programs.

Without the management models, cache misses due to initial loads are unavoidable, while the number of misses resulting from premature replacements is sensitive to the replacement policy, the accessing pattern, and the data packing. However, as we described in chapter three, both types of cache misses could be reduced by the management models through prompting and allocation.

Assuming that we have only two types of variables: scalars and array variables, then for either type there is a different data packing procedure. For scalars the aim of a good data packing procedure is to pack variables that tend to be accessed together within the same basic block. For array variables a good data packing procedure avoids the possibility of a contention by allocating arrays that are accessed together to locations which map to different cache classes.

Throughout the program execution, scalar variables remain within the same block (class), while the elements of the array variables sweep through the cache. Under such circumstances any block containing scalar variables may be prematurely replaced by blocks containing array variables that map to the same class. Furthermore, if any of
the scalar variables is accessed at the same time with the arrays that replaces their block, in a loop, then a large number of premature replacements will occur.

**Example**

```
DO 10 I = 1 TO N
   T = X(I) + Y(I)
   S = S + Z(I) * W(I)
   R(I) = T + W(I)
10 CONTINUE
```

Assuming a set associative cache with two blocks to a class, then contention would arise if more than two array elements are mapped to the same class. Moreover, if we assume that we are accessing the scalars from the cache in every iteration, then contention between the block holding the scalars and blocks belonging to an array element may occur. As execution proceeds the elements of the array will move from one cache class to the next in sequence. In other words, arrays sweep through the cache, while scalars remain within the same class throughout the execution of the program. Therefore, if a block containing scalar elements is in the path of an array variable then the possibility of contention is very high. []

The problem depicted in the previous example may be considered as a major cause of premature replacements under a set associative cache structure. In the following section we shall show how this type of problem could be tackled by the compiler without resorting to any
hardware modifications.

5.2. Optimization For a Transparent Cache

Our management models do three tasks: packing, allocation, and deallocation of cache blocks. All these functions are done with the knowledge of the cache structure, and the availability of special hardware functions that are required to facilitate the implementation of the models.

In this section we shall present some optimization the compiler could do without the need of the special hardware functions. Under such an assumption the compiler has a limited number of optimization techniques, in particular, block packing for scalars, and storage assignment for the blocks of arrays. With the former we try to minimize the number of blocks in the cache, especially during loop execution; while the aim of doing storage assignment for arrays is to reduce the possibility of contention for arrays accessed within the same loop. This makes loops (cycles) and their detection a vital source for optimization in cache systems.

Do loops, and similar other structured loop constructs are easily detected by the compiler. Cycles that are a result of the GOTO statements are sometimes hard to detect. Such cycles are called natural loops. Algorithms for detecting natural loops have been given by Baer and Caughey[72], and Aho and Ullman [79].
5.2.1. Loop Structured Packing for Scalars

In the following algorithm we shall make use of Ullman's ideas for detecting loops, and put to use the packing algorithm suggested by Baer and Caughey. This algorithm has been used for the packing of code rather than data. Nonetheless, we shall modify it for the packing of data.
Algorithm A

A1  Build the flow graph $G(V,E)$.

A2  Find all natural loops.

A3  Construct a dag $H(U,F)$ such that $U$ represent the set of natural loops, and all other loop constructs. If all the blocks of $u_i$ are in usbi, i.e we have a loop nesting, then $(u_i, u_j)$ is in $F$.

A4  Consider the vertices of $H$ that have no predecessor to be roots. Also assume that the immediate successors of a vertex are its sons, its immediate predecessor will be its fathers. With these assumptions the depth $D_i$ of a vertex $u_i$ is the length of the longest path from a root to $u_i$. Using topological sort we can compute the depth value for all the vertices. A root will have depth of one all blocks in the original flow graph that are not in $H$ will have a depth of zero.

A5  Build a proximity matrix for the scalar variables of the program. The entries of the matrix are computed as follows:

$\rightarrow$ Assign to each block $i$ of the flow graph the value $10$ raised to the power $D_i$, its depth.

$\rightarrow$ Two variables that are within the same block will get the value of the block added to their corresponding entry in the matrix.

A6  Apply the modified Single-link algorithm as suggested in the previous chapter. If a cluster reaches a cache block size, then
remove it from the proximity matrix.

**Analysis of Algorithm A**

Almost each step of algorithm A is an algorithm by itself. Step A1, the building of a flow graph, is covered in [Aho and Ullman '79] and [Kennedy '80], the complexity of this algorithm (A1) is linear in the number of statements in the program. Since the flow graph is also built for various other optimizations, we shall assume that it is given and thus its complexity will not be included in the complexity computation for algorithm A.

The algorithm for step A2 is also given in [Aho and Ullman '79]. The detection of natural loops depends on the detection of back edges, those edges in the flow graph whose heads dominates their tails. The complexity of the algorithm that finds back edges is $O(n + e)$. Once a back edge is found, all the blocks belonging to the natural loop are determined in time linear to the number of blocks that are predecessor to the tail of the back edge. Since there is a constant number of natural loops within a program each requiring a linear time, the detection of all natural loops is linearly bounded by a constant times the number of back edges. The constant here is less than the number of basic blocks for the program.

Once all natural and structured loop are found a simple binary matrix is built to reflect the block membership for each cycle(loop). Cycle embeddedness is determined by taking the intersection (AND) of the rows. A cycle $C_i$ is in embedded in $C_j$ if $C_i \land C_j$ is equal to $C_i$. 
In this case the elements \((j,i)=1\) and \((i,j)=0\) of the matrix representation of the dag \(H(U,F)\). Thus the construction of \(H\), the complexity of \(A3\), is of order \(L^2\) where \(L\) is the number of loops.

The complexity of \(A4\) is equal to that of topological sort (breadth first search), which is \(O(n + e)\). Where \(n\) is the number of vertices (basic blocks) and \(e\) is the number of edges. This value is bounded by \(O(n^2)\) [Baer]. This algorithm was first suggested by [Baer and Caughey '72], but was not implemented:

"A sort on levels 1, which satisfies this criterion, is rejected because we want to keep as much as possible the image of the structure of the source program."

In our case we are working with the packing of data elements, and thus we need not follow the other algorithm suggested by [Baer and Caughey '72], which also happen to be of \(O(n^3)\).

Since the order of traversal in \(A5\) is not important breadth first search could be used. Thus we can use the same algorithm as step \(A4\). However, the value of \(n\) and \(e\) are larger now because we applying the algorithm for the whole graph.

The complexity of \(A6\) is that of the Single-link, which is of \(O(m^2)\), where \(m\) is the number of scalar variables in the program.

The overall complexity is \(O(n+e) + O(L^2) + O(m^2)\). Generally for a given program the number of variables \((m)\) are much larger than the number of basic blocks \((n)\) in the flow graph of the same program \((m>>n)\). In this case the complexity of algorithm \(A\) is \(O(m^2)\).
5.2.2. Array Packing

Heuristic approach to the problem of storage optimization for arrays was first suggested and implemented in Alpha compiler [Yershov '71]. Yershov approach was to do a live analysis of the arrays then build a conflict graph. A set of possible overlay is then determined using a combination of coloring and packing heuristics.

Fabri['79] fine tuned Yershov's approach and introduced various heuristics of her own. Abu-Sufah['79] looked at the problem with virtual memory in mind. His aim was to minimize the number of pages allocated for arrays. Since, in most cases, array accesses occur within loops, Abu-Sufah suggested various loop transformations that would allow the utilization of a small number of pages for a number of large arrays. In Fabri's and Yershov's approaches the array addresses are determined at compile time so that some arrays will have the same main memory addresses. This means not all of the arrays will be in main memory at any moment, and in Abu-Sufah's approach only a number of pages for each of the arrays that are being accessed at the time (live). In our approach we would like to minimize cache usage but would like to have as much as possible of the arrays to be in main memory so that a miss would not lead to a page fault. With this assumption we shall not apply, as given, any of the three methods but will devise one that would utilize the underlying concepts of all of three approaches.

Our main objective is to have arrays that are in conflict, appear in the same loop, be in different cache classes. Hence avoid
the possibility of contention. This is accomplished by allocating the addresses of arrays in conflict so as not to map to the same cache class. In other words arrays, the Mod function of the addresses of arrays that are not in conflict are equal, while those in conflict have different values.

Thus array allocation involves three steps. First, we build a conflict graph with vertices representing the array names, and an edge is drawn between two vertices if their corresponding arrays are within the same loop structure. Our second step would then color the graph [Chaitin et al '80]. Finally, arrays that are not in conflict, i.e., have the same color, could be mapped to the same cache class and are assigned storage with this strategy in mind. On the other hand, arrays that are in conflict, have different colors, must reside in different cache classes during the program execution. In this scheme we assumed that conflict may arise at the same relative location, i.e., A(1) is in conflict with B(1) and not B(17). Such an assumption simplifies the problem, but in reality this might not hold true. To avert this situation, we first assign cache blocks for the arrays that start with the initial value of the index variable, then assign blocks for the arrays in the order, ascending order, they are away for the initial index value. The following example illustrate our point.
Example

Given a two way set associative cache, with 8 words per block:

\[
\text{DO 10 I = 1 TO N} \\
A(I) = B(I+8) + C(I+16) \\
10 \text{ CONTINUE}
\]

The three arrays \(A, B,\) and \(C\) are in conflict. If we assign \(C\) to class 1, \(B\) to class 2, and \(A\) to class 3, then all initial accesses will be from class 3 and the problem of contention would remain throughout the loop execution; thus leading to extensive cache misses. On the other hand, if we assign \(A\) to class 1, \(B\) to class 2, and \(C\) to class 3, then the initial accesses would be from the classes 1, 3 and 5 respectively. Thus the problem of contention is less likely to occur.[]

In the previous example we tackled the problem of contention for the single loop. If, for example, array \(D\) is in conflict with array \(C\) in another loop, then array \(D\) will be assigned to class 4. However, if the conflict between \(C\) and \(D\) is like the conflict between \(A\) and \(B\), e.g. \(C(I+8) = D(I)\), then all accesses for \(C\) and \(D\) will be from the same class.

To forestall such problems, some form of ordering must be built within the conflict graph. The implicit type of ordering could be achieved through the construction of a directed conflict graph. However, the construction of directed conflict graph might be more
complex than the construction of an undirected conflict graph. To reduce the complexity of this problem we built two conflict graphs. An undirected graph will be built for those arrays which do not have an offset value other than that of the loop index, and a directed graph of the remaining arrays.

When coloring, each graph will be colored with a disjoint set of colors. Therefore, arrays that are in conflict but have been placed in different graphs would still map to different cache classes. Allocation is first made for the arrays in the undirected graph, followed by the allocation for the arrays in the directed graph.

The previous scheme may simplify the problem but would lead to the usage of larger number of colors. In other words, for the solution of this problem we shall be using more cache classes, which is contrary to the intent of most coloring algorithm. Nevertheless, the purpose of coloring here is for allocation and not for the minimal allocation as generally aspired by many register allocation coloring algorithms [Chaitin et al '80]. With register allocation the number of registers, 16 for the IBM 370, is possibly much less than the number of variables that may be in conflict, while the number of arrays that are in conflict would rarely exceed the number of cache classes. For example, the IBM 370 cache has 256 classes (sets). Thus if the number of arrays (vertices) that are in conflict is less than the number of cache classes available for arrays, then any coloring would suffice. The simplest solution in this case is to assign each vertex a unique color. Nevertheless, there is always the desire to
use less colors than the number of arrays, which in this case may reduce the affect due the sweeping property of arrays. A simple algorithm by Carre[79] produces all possible colorings for a given graph. Here two nodes having no edge between them (not in conflict) are coalesced if assigned the same color or have an edge drawn between them if assigned different colors. In the process if the graph becomes a complete graph then coloring is complete and the number of nodes in the resulting graph is equal to the number of colors for the original graph. The basic idea here is that complete graphs have only one possible coloring.

The following algorithm gives the memory allocation that avoids cache contention due to arrays. For reason mentioned earlier we shall build two conflict graphs. However, since the building and the coloring of a directed graph will not be discussed in this thesis, we shall only work on the coloring of the undirected graph. Furthermore, we shall assume that the number of classes in the cache is much larger than the number of arrays in conflict. Nonetheless, if this case does not hold then we can apply one of the graph coloring algorithms that uses recursive backtracking so as to get a coloring less or equal to the number of cache classes. The complexity of such algorithms is $O(mn)$ where $m$ is the number of cache classes or available colors[Borowitz and Sahni '78].

In case the number of arrays that may be in conflict is larger than the available cache classes, we could apply horizontal distribution to reduce this number. With this in mind we could apply the a
simpler version of Carre's algorithm [Carre '79]. Here we only coalesce nodes with no edge between them until we arrive at a complete graph. Since this algorithm is polynomial in time and the problem of finding a chromatic number is an NP-complete problem, this algorithm may not result in the chromatic number for the graph, but will generally result with a number of colors less than the number of nodes (arrays). The reason here is that with coalescing the number of colors required is reduced.

**Algorithm B**

**B1** Build a conflict graph $G(V,E)$ for the arrays of the program with the set of vertices representing the array names. An edge is drawn between two vertices if the arrays are within the same loop (cycle).

**B2.** Color the vertices of the graph such that two vertices connected by an edge must have different colors (this is called graph coloring). We shall use the coloring algorithm as given by [Carre '79], except that we only keep coalescing the vertices that are not in conflict until we end up with a complete graph.

**B3.** Assign nodes (arrays) with the same color the same memory location, so that they will map to the same cache class.

**Analysis of Algorithm B**

Building the conflict graph is $O(n + e)$, where $n$ is the number of arrays (vertices) in the program and $e$ is the number of edges drawn. In step B2 we coalesce two nodes then check for a complete graph. If
the graph is represented by a binary matrix then checking for a complete graph requires the application of the logical and on the rows, then anding the elements of the resulting vector. A value of one means that the graph is complete. Thus checking for graph completeness is \(O(n)\). Since the number of nodes that get coalesced is less or equal to \(n\), the complexity of this step is \(O(n)\).

Step B3 is simply \(O(n)\), and thus the overall complexity of algorithm B is \(O(n^2)\).

As mentioned earlier, to reduce the number of arrays that are in conflict we can apply the horizontal distribution. This would also simplify the mapping problem.

To take scalars into consideration here, the blocks containing scalars must be assigned addresses in such a way that the arrays they are in conflict with would never over write them. Such a conflict arises when a scalar from the block and the array are in the same loop. As we mentioned earlier this conflict leads to contention. One way of avoiding this problem is by assigning separate storage area at the top of the cache for the blocks of scalars. This way the arrays must be larger than the cache size to over write a block of scalar variables. This case would hold true if there are few scalar blocks and the arrays' sizes are less than the cache size.

The previous ideas may enhance the cache performance. Other approaches that we do not expect to have as significant an effect are discussed in the following paragraph.
Since all modified blocks have to be copied back to main memory even those containing only temporaries, the need to separate modified variables into different blocks according to type ceases to exist. The new aim would then be to reduce the number of modified blocks by packing only variables that will be modified. As such when doing the packing, we separate the variables into two types: those that will be modified and program constants.

Other optimization ideas may include the movement of the initialization statement of a variable to where the variable is first accessed. This is particularly important for array variables.

5.3. Heuristics for The Prompting Model

We say a variable is defined if it is on the left-hand side of an assignment, a parameter in a call statement, or an element in a read statement. On the other hand, we say a variable is used if it is on the right-hand side of an assignment, a parameter in a return statement, or an element in a write statement.

The implementation of the prompting model requires the information as to where variables are first defined, in which blocks they are used or redefined, and after what point in the program will they no longer be accessed. Such information is needed for prompting, packing, and releasing of cache blocks respectively.

Variables defined in the same basic block are generally used in different basic blocks. Similarly variables used in the same basic block are usually defined in different basic blocks. Packing vari-
ables that are defined or used in the same group of basic blocks would generally increase the packing factor and the block hit ratio, both of which are essential in reducing the miss ratio. For our model, such packing may require less prompting of cache blocks.

First the variables are classified according to their type: constants, computed variables, input variables, or temporaries. Secondly, each of the classified group of variables are packed into cache blocks in a way that maximizes the packing factor. In other words, any group of variables that tend to be accessed together, regardless of the sequence of access, need to be within the same cache block. For example temporary variables within a basic block may be packed into the same cache block. More generally, we shall use the packing algorithm used in the previous section (algorithm A) but for each of the variable types.

Besides packing the model has to decide where to place the prompting requests. Since variables have to be prompted before their use, prompting requests must be placed in locations that are in the path of the first use of the variables. This accomplished by placing the prompt instructions at one of the blocks that dominates the block containing the variable to be prompted.
Definition

For a given flow graph a node \( m \) dominates a node \( n \), if every path from the initial node of the flow graph to \( n \) passes through \( m \).

The dominance is a reflexive, transitive, and antisymmetric relation.

Example

The initial node is \( A \)

\( B \) dominates \( B \)
\( D \) dominates \( D \)
\( C \) dominates \( \{D, C\} \)

Graph 2

\( A \) dominates \( \{A, B, C\} \) but because of transitivity \( A \) also dominates \( D \). Thus \( A \) dominates \( \{A, C, B, D\} \).

The following algorithm will decide the location of the prompting requests for the blocks of scalars. First scalars are packed into cache blocks (SCB). In the flow graph each variable is then substituted with the name of the block it has been packed into. The algorithm then determines the set of SCBs that are accessed in each node of the flow graph. Prompt requests for the SCBs are then placed at one of the dominators of each of the nodes. To avoid unnecessary prompts, all prompt requests of the SCBs should be placed outside
loops. Since these requests are made in a dominator of the node accessing an SCB, all the dominators of a node that are within a loop will not be considered.

Release and transfer requests for the blocks of temporaries and computed variables will be covered in another algorithm.
ALGORITHM C

C1 Do the scalar packing(SCB) by implementing algorithm A.

C2 Replace each variable in the flow graph with its corresponding SCB name. Then determine the set of SCBs for each node(basic block) in flow graph.

C3 For each node in the flow graph find its dominator. Then eliminate all the dominators that are within a loop or a cycle. This way prompts for the SCBs will be outside cycles to avoid unnecessary prompts.

C4 For each node find the difference set, of the set of SCBs accessed in that node with the set SCBs accessed in each of its dominators as computed in C2. The dominators that give rise to non-empty sets are selected.

C5 Order the dominators selected in C4 with increasing distance from the dominant node.

C6 Select the first dominator with non-empty set, then place a prompt instruction for all the set of SCBs at the beginning of that dominator.

C7 Eliminate all SCBs prompted from the sets of SCBs found in the other dominators. If all the sets of SCBs to each dominator is empty then done else go to C6.

C8 For each of the nodes eliminate all but one of the multiple copies of the same prompting requests.
Analysis of Algorithm C

Step C1 has the complexity of algorithm A. Step C2 is of $O(n)$ the number of nodes in the flow graph. Aho and Ullman [79] give a simple iterative algorithm for computing the dominators to every node in the flow graph. The algorithm converges very quickly in several steps. If the algorithms has the depth first ordering (DFO) for the flow graph and back edges and backward edges are the same, then the set of dominators could be computed in the first pass through the program [Aho and Ullman '79]. Generally, the Ullman algorithm is $O(n+e)$. The determination of which dominator is within a loop could be taken from algorithm A or simply computed and that also is $O(n+e)$.

If a dominator has all the SCBs of a node, i.e. the difference set is empty, then prompting at this dominator would minimize the gain. On the other hand, if the difference of sets is not empty, then some of the SCBs are not accessed at the dominator but will be accessed at the node, then prompting for all these SCBs (the result of the set difference) would result in a performance gain.

The difference between the sets $A$ and $B$ is simply equal to the intersection of $A$ with the complement of $B$. By using bit vectors to represent the SCB set for each node we can do step C4 in time $O(n)$. 
Steps C5 through C8 would also take time \( O(n) \). Step C8 is added to insure prompting is done for all the SCB's. The complexity of the algorithm, discounting C1 and C2, is \( O(n^e) \).

With the blocks for scalars prompted, we now turn to an important feature of the prompting model: the releasing of scalar blocks. Here we assume the existence of a basic block in the flow graph such that all exists from the program must go through. In other words it is the dual of the entry node. Thus for the block release algorithm we simply reverse the arrows in the flow graph then apply algorithm C, but use release requests rather than placing prompt requests.

**Algorithm D**

D0  Reverse the arrows in the flow graph.

D1  Apply algorithm C but use the release instruction rather than the prompt instruction.

**Analysis of Algorithm D**

Since algorithm D is a dual of algorithm C which uses the steps, it thus have the same complexity of C.

Earlier in the thesis we recommended that prompting of cache blocks should be done in parallel with execution and when the cache main memory bus is free. This would likely happen during loop execution. Therefore, data needed after the loop should be requested before commencing the execution of the loop. This raises the question of when do we prompt for data within the loop.
Arrays that span several cache blocks are generally accessed within loops. Each prompt request is typically for one cache block. Therefore, array variables may require a number of prompt requests. Such requests cannot be made outside the loop, because this may require the use of several prompt instructions, which reduces the possibility of satisfying all of these requests. Moreover, prompted array elements may replace valuable information before these array elements are ever accessed. Furthermore, the number of the array element that get accessed within a loop might not be known prior the loop, and is only determined during the execution of the loop itself.

An alternative approach is to do array prompting during the execution of the loop. This could be accomplished by first applying vertical distribution, then placing the prompt between the distributed loops. There are basically two possible ways for placing the prompt requests.

In the first method we place prompt requests for arrays accessed in a statement before the execution of the loop containing the preceding statement. In the second method we place prompt requests for arrays accessed in a statement just before the loop containing that statement.

Example

```
DO 10 I = 1, N

A(I) = C(I) + D(I)

D(I) = D(I) + B(I)
```

10 CONTINUE

CASE 1

Prompt (A(1), 0, 1)
Prompt (C(1), 1, 0)
Prompt (D(1), 1, 1)
Prompt (B(1), 1, 0)

......

DO 10 IP = 1, N/R
    ILB = 1 + (IP-1)*R
    IUB = IP*R
    Prompt (B(ILB), 1, 0)
DO 20 I = ILB, IUB
    A(I) = C(I) + D(I)

20 CONTINUE

Prompt (A(IUB+1), 0, 1)
Prompt (C(IUB+1), 1, 0)
Prompt (D(IUB+1), 1, 1)
DO 30 I = ILB, IUB
    D(I) = D(I) + B(I)

30 CONTINUE

Release (A(ILB), 1)
Release (D(ILB), 1)

10 CONTINUE

ILB = IUB + 1

CASE 2

Prompt (A(1), 0, 1)
Prompt (C(1), 1, 0)
Prompt (D(1), 1, 1)

......

DO 10 IP = 1, N/R
    ILB = 1 + (IP-1)*R
    IUB = IP*R
    Prompt (A(IUB+1), 0, 1)
DO 20 I = IUB, IUB
    Prompt (C(IUB+1), 1, 0)

20 CONTINUE

A(I) = C(I) + D(I)

Prompt (B(IUB+1), 1, 0)

DO 30 I = ILB, IUB
    D(I) = D(I) - B(I)

30 CONTINUE

Release (A(ILB), 1)
Release (D(ILB), 1)

10 CONTINUE

DO 40 I = IUB+1, N
Prompt (B(ILB),1,0)
A(I) = C(I) + D(I)
DO 40 I = ILB, N
   A(I) = C(I) + D(I)
   D(I) = D(I) + B(I)
40 CONTINUE
40 CONTINUE
Release (A(IUB),1)
Release (D(IUB),1)
DO 50 I = ILB, N
   D(I) = D(I) + B(I)
50 CONTINUE
Release (A(ILB),1)
Release (D(ILB),1)

The previous example illustrates two ways for placing prompt requests within a distributed loop. Both methods increase the size of the programs code. Nevertheless, these methods are used for scientific programs where code size is only a fraction of the data size, and thus the increase in code would probably amount to a small increase in the memory requirement for the program.

In the prompting model we recommend the implementation of case 1. In this method only the initial block of arrays referenced in the first statement of the loop are requested before loop entry. Thus the number of blocks in the cache that are prompted but not yet referenced is reduced. This in turn reduces the possibility of premature replacement.

In the following algorithm we shall describe the insertion of the prompt instruction within the program's code.
ALGORITHM E

E0. Apply algorithm B to determine the addresses for the arrays.

E1. Do a vertical distribution for the loop.

E2. Number the statements in the loop. For each statement find the set of arrays accessed.

E3. For \(i = 1\) to \(n - 1\) do E4 and E5 (\(n = \# \text{ statements}\)).

E4. Take the set difference between the set of arrays in statements \(i + 1\) and in the union of sets in statements 1 to \(i\); call this set \(j\).

E5. Before the loop containing statement \(i\) place the needed prompt requests for the loop containing statement \(i + 1\), with the address starting at ILB. If there is an array in the left hand side of statement \(i + 1\) and is in set \(j\) then do a prompt request (a non-load), similarly for all temporary arrays in the set \(j\). For all other arrays in the set \(j\) do a prompt request (a load).

E6. Before the loop containing statement \(n\) place a prompt request (a non-load) for all the temporary arrays in first statement, and for the array appearing on the left hand side; all the prompts must be starting at the address IUB. For all the rest of the arrays do a prompt request (a load), again starting at the address IUB.

E7. For all the computed, or temporary arrays do a release request after the loop where they were last accessed.
E8. After the last statement in the loop, make a copy of the loop with all the prompt requests except for the prompts introduced in E6.

Analysis of Algorithm E

In this algorithm we do vertical distribution and place prompt requests. The complexity of vertical distribution is $O(n)$, where $n$ is the number of statement within the loop. On the other hand, the placing of prompt requests is linearly dependent on the number of arrays in the loop, i.e. $O(m)$ where $m$ is the number of arrays in the loop. Thus the overall complexity of algorithm E is $O(n+m)$.

5.4. Heuristics for the Explicit Management Model

In the explicit model the compiler carries more of the burden of the cache management than its counterpart, the prompting model. Some of the functions performed by this model are, to a certain extent, an exact copy of those done by the prompting model. In particular, those functions involved in the packing of the scalars into cache blocks. Moreover, we can also apply the prompt algorithm (algorithm C) to determine where to place the load requests for the packed blocks.

In the prompting model we tried to avoid contention between arrays in the same loop. Furthermore, conflicts between cache blocks belonging to arrays and cache blocks holding scalars have been minimized. Nonetheless, if such conflicts do arise, in the prompting model, they will lead to a performance loss. On the other hand, such conflicts are intolerable in the explicit model and would definitely
cause program errors. In addition, array conflicts would also lead to program error. In this section we shall propose a heuristic implementation of the explicit model which forestalls any anticipated conflicts that might be a source of program errors.

In the explicit model each array variable is allocated to fixed number of cache blocks. Arrays that are in conflict will have their blocks in disjoint cache classes, thereby forestalling any potential contentions. Each array would then cycle through the blocks it has been assigned to, rather than sweep through the cache as in the prompting model. In other words, the prompted (loaded) array blocks would replace only one of the blocks allocated to the same array. Since arrays are usually accessed in a fixed sequence, the FIFO replacement policy is selected. To automate the replacement decision the special index register, presented in chapter three, is used.

An array may have a contention with itself. Such contention occur for arrays that are the cause of dependency relation between two statements within the same loop. If the threshold of the dependency is larger than the number of blocks assigned to the array then program error would arise. This is because the CPU will be trying to write an array element to a block not assigned to the array, or access an array element which has not been loaded in the cache or that has just been overwritten. The former come about with true dependence having a threshold larger than the number of blocks assigned. The latter occur during antidependency with similar threshold. To remedy this problem arrays that are part of a dependency
relation are allocated a number of blocks that exceeds the threshold of the dependency. Other arrays are assigned an equal number of fixed blocks. This makes dependency analysis vital for the implementation of the explicit model.

For the explicit model we have two main algorithms. In the first, we determine the number of blocks needed for each array. The second algorithm places the load requests for array elements within a loop. The algorithms for scalar packing, and block prefetching are similar to those given in the prompting model, except for minor modifications. Thus only the array related algorithms are presented here.
**Algorithm F**

F1. Do a dependency analysis and find the threshold of each dependency and its corresponding array.

F2. Build a conflict graph for the arrays of the program that are not part of a dependency or whose threshold do not extend further than two cache blocks.

F3. Allocate each of the arrays of F2 two blocks. Arrays that got the same color in the graph are assigned to the same two blocks. If all the arrays of the program have been considered in this step then STOP.

F4. Build a conflict graph for the arrays having threshold that extends further than two cache blocks. Place the actual threshold value on each node. If an array is involved in a more than one dependency, place the largest value on the corresponding node.

F5. Color the conflict graph of F4 starting with the nodes with highest value.

F6. For each color allocate a number of blocks equal to the highest value node in that color. Arrays having the same color are assigned the same blocks.

**Analysis of Algorithm F**

In this algorithm we do dependency analysis and graph coloring. If the number of arrays that are in conflict is small then we apply the linear alternative of algorithm E. In this case E would be a
linear algorithm. If the number of arrays that are in conflict is larger than the cache size, then we first try to reduce this number by horizontal distribution. If however, horizontal distribution could not be applied or if applied does not reduce the number of arrays in conflict then we use the recursive backtracking method [Horowitz and Sahni '78]. The complexity of E would then be $O(n^m^2)$.

In the explicit model we allocate a number of blocks for the temporary arrays. However, we never do a release request, but rather reuse them by other arrays allocated to the same blocks. Similarly for write only blocks and array elements that appear on the left hand side of an expression; since space for these arrays have already been allocated. This affect is clearly seen in the following algorithm which similar to that of algorithm F.

In algorithm G we shall apply case 2 for the load requests. Here load requests for arrays referenced in a statement are placed before the loop containing that statement. Since space for the arrays is allocated at compile time, the problem of premature replacement will not arise.
**Algorithm G**

G0. Apply algorithm E to determine the addresses for the arrays.

G1. Do a vertical distribution for the loop.

G2. Number the statements in the loop. For each statement find the set of arrays accessed.

G3. For i=1 to n do G4 (n = # statements).

G4. If the set of arrays for statement i is not empty then, before the loop containing statement i place the needed loading requests for the arrays in the set, with the address starting at IUB. Eliminate the arrays requested from the set of arrays of statements i+1 to n.

G5. For all the arrays computed do a transferind request after the loop where they were last accessed with the address ILB.

G6. After the last statement in the loop, make a copy of the original loop, but with the limits of IUB+1 to N. Place all the transferind requests of G5 after this loop.

**Analysis of Algorithm G**

Although this algorithm and algorithm E are slightly different, they have the same complexity. In other words the complexity of G is O(n+m), where n is the number of statements in the loop, and m is the number of arrays referenced in the loop.
CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

In this dissertation, I proposed a number of software enhancement methods for the cache. I shall now recapitulate my contribution with recommendations for future directions.

First, the three main memory enhancement models have been modified and grouped under a single enhancement model, tailored for the cache structure. In particular, for main memory no attention has been given to scalar packing since they usually make up a fraction of a page size. Nonetheless, in the cache scalars may require a number of cache blocks and their packing may affect the cache performance. For example, scalar and array variables have been classified and packed into four classes so as to reduce block traffic in the cache main memory bus.

Theoretical analysis of the enhancement methods has been undertaken. Here we showed that optimal packing is hard problem, but heuristics could be utilized in achieving a better performance through packing. We then analyzed the conditions needed for performing loop distributions, and proved that only one condition needed to be satisfied for performing these distributions. Previously the compiler had to look for two types of dependencies, normal and loop carried, then look for the occurrence of a recurrence relation, the compiler would then try to break this recurrence by eliminating one of
the dependencies in the recurrence. Now the compiler simply looks for carried dependencies and try to break it so as to perform the various loop distributions.

Other contributions included the investigations of various causes for the degradation of cache performance, and then offering a possible remedy to these problems. In particular we realized the affects and causes of contention in a set associative cache, we then provided various heuristic methods for avoiding this problem. For example, arrays that are referenced within the same loop are mapped to different cache classes. An alternative solution would be to do horizontal distribution, the arrays would then be in different loops.

A major extension to this work would be to incorporate some of this ideas within the loader program. This way contention due to subroutine calls would be avoided. Another future directions would be for the compiler to pass some of data accumulated about possible program behavior in the cache to operating system which in turn would try to minimize premature replacements due user-initiated supervisor calls (SVCs).

Finally, we hope to extend our work in the near future with some empirical results by performing a simulation of our models. Such work may be done with the technical support of IBM and the financial support of the Saudi Arabian National Center of Science and Technology (SANCST).
APPENDIX

A simple description of flow analysis has been provided by Muchnick and Jones[81]:

"Flow analysis is a tool for discovering properties of the run-time behavior of a program without actually running it. The properties discovered usually apply to all sequences of control and data flow, and so give global information impossible to obtain by individual runs or by inspection of only a part of the program."

Flow analysis is comprised of several steps. First the program is divided into basic blocks. A flow graph is then built with nodes corresponding to the basic blocks. The set of edges for the graph reflect the flow of control between the blocks. Construction of the graph depends on the recognition of the basic blocks.

For a given program, any sequence of statements that will be executed in the order given form a basic block. A basic block is only accessible through its first statement. This statement is referred to as the block leader.

The first statement in the program, and all statements that are the target of a conditional or unconditional 'goto' are block leaders. Furthermore, a statement following a conditional 'goto' is also a block leader. Thus a basic block consists of a leader followed by a sequence of statements up to but not including the next leader. This makes the recognition of basic blocks and the construction of the flow graph depend on the discovery of block leaders. Details of these algorithms are given in a survey paper by Kennedy [81].
In this paper, Kennedy classifies data flow problems as either forward, or backward flow problems. For a certain point P in a given program, the forward flow problems relate to definitions which affect computation at P, while the backward flow problems address the question of what happens after control leaves P. Solving either type of problem may require the testing of an infinite set of paths. Flow problems, however, can be formulated into a set of data flow equations. Such a formulation reduces the testing to only a finite set of paths. The construction and the solution of such data flow equations make up a major part of data flow analysis.

For every flow problem there are two types of equations: transfer and confluence rules. Transfer rules describe data behavior in going from one block to the next in the flow graph. Confluence rules, on the other hand, describe what happens when paths meet. Information gathered using data flow analysis is often used in compilers to aid optimization.
BIBLIOGRAPHY


Baer and Caughey. (1972) "Segmentation and Optimization of Programs from Cyclic Analysis."


 rotated


خلاصة رسالة الدكتوراه
إدارة المترجم للذاكرة الخفية

ينتظر جهاز الكمبيوتر المكالما ذو القدرات العالية على معالج سريع إلى جانب ذاكرة تماثيله في السرعة وضع عدة ملايين رمز (حرف أو رقم). ولكن الأسباب الاقتصادية الحالية حالت من وجود كمبيوتر بذاكرة ضخمة تماثل المعالج في السرعة، مما دعا مماشي آخرين الكمبيوتر إلى اعتماد الذاكرة الخفية (الكاش). إن مسرح ذاكرة الخفية وفاعليتها في التوافق بين سرعات المعالج والذاكرة، جعل استخدامها جزءًا أساسيًا في اجراء الكمبيوتر ذو
القدرات العالية.

إن تحسين عمل الذاكرة الخفية برهن أهميته في تطوير اجراء الكمبيوتر في معظم الحالات ويمكن تصنيف التحسينات إلى فئتين: آلية أو وسيلة. ويمكن القول أن التحسينات المماثلة التي سبطنت فعاليتها للذاكرة الرئيسية كتبت
فعاليتها للذاكرة الخفية، من أهم أسباب ذلك فعالية الذاكرة الخفية للبرامج أولاً وسرعة استجابة الذاكرة الرئيسية كثيرة هذا يؤدي إلى استخدام التحسينات
العليه في تطوير عمل الذاكرة الخفية.

إن التطويرات التي طرأت على تحسين فعالية البرامج وقبل المترجم
اللغات أبيضت جدارتها في قيم وتحسين سير البرنامج (الذي ترمج). من أفضل الوسائل لتحسين عمل البرنامج هي التي شاخص ببعين الادعاء مكونات الجهاز إلى جانب معلومات عن كيفية سير البرنامج، وعلى هذا الأساس اقترحنا
نموذج إدارة الذاكرة الخفية: النموذج الحاث، والنموذج ذو التدبير الين.
كل النموذجين يعتمدان على نفس مبادئ ثلاثة تحسينات: الطاب المسبق
- والعملية وتحويل الحلقات (التي هي جزء من البرنامج). هذه التحسينات برهنت
أهميةها في تطوير عمل الذاكرة الرئيسية، بعد تحليلنا لتلك الطرق ومحاولة
تطبيقها في الذاكرة الخفية وجدنا أن الطريقة الملائمة لتعمل البيانات يصعب
تطبيقها، مع ذلك حاولنا إيجاد طريقة تنفيذية للتعامل، ثم وضعنا عدة شروط
لتحويل الحلقات الهدف الأساسي لهذه التحويلات هو تسهيل عملية الطلب المسبق للبيانات خلال تنفيذ الحلقات.

لكل من النموذجين يضع المترجم الطلب المسبق داخل البرنامج، هذه الطلبات تلبى خلال تنفيذ البرنامج، في حالة النموذج البيئ تستنتج قرارات الاستبدال عند الترجمة، أما بالنسبة للنموذج الحاث فتجري وتنفذ أحياناً لهذا النموذج اقتراحًا وجود رقم ثاني (بت) لكل قالب داخل الذاكرة الخفية حتى يسهل عملية الاستبدال.

معالجة البيانات المتجمعة (المجدولة) اتخذت بين الاعتبار في هذه الرسالة في النموذج البيئ استخدمنا طريقة فهرسية للتحكم على توصيل البيانات المجدولة في الذاكرة الخفية، في هذه الحالة يكون الطلب من الدائرة الرئيسية عند الحاجة إلى قالب من البيانات، أما في باقي الحالات فتطلب البيانات من الذاكرة الخفية.