An Accurate Prefetching Policy for Object Oriented Systems

by

Dong Ho Song

Ph.D. Thesis

December 1990

The University of Newcastle upon Tyne
Computing Laboratory

Abstract

In the latest high–performance computers, there is a growing requirement for accurate prefetching(AP) methodologies for advanced object management schemes in virtual memory and migration systems. The major issue for achieving this goal is that of finding a simple way of accurately predicting the objects that will be referenced in the near future and to group them so as to allow them to be fetched same time. The basic notion of AP involves building a relationship for logically grouping related objects and prefetching them, rather than using their physical grouping and it relies on demand fetching such as is done in existing restructuring or grouping schemes. By this, AP tries to overcome some of the shortcomings posed by physical grouping methods.

Prefetching also makes use of the properties of object oriented languages to build inter and intra object relationships as a means of logical grouping. This thesis describes how this relationship can be established at compile time and how it can be used for accurate object prefetching in virtual memory systems. In addition, AP performs control flow and data dependency analysis to reinforce the relationships and to find the dependencies of a program. The user program is decomposed into *prefetching blocks* which contain all the information needed for block prefetching such as long branches and function calls at major branch points.

The proposed prefetching scheme is implemented by extending a C++ compiler and evaluated on a virtual memory simulator. The results show a significant reduction both in the number of page fault and memory pollution. In particular, AP can suppress many page faults that occur during transition phases which are unmanageable by other ways of fetching. AP can be applied to a local and distributed virtual memory system so as to reduce the fault rate by fetching groups of objects at the same time and consequently lessening operating system overheads.

Acknowledgements

Foremostly I owe a debt of gratitude to my supervisor Dr. Lindsay Marshall for his help and encouragement, and in particular, for suggesting this research area. His reading and commenting upon the numerous drafts of this thesis have been invaluable in completing this thesis. His efforts are greatly appreciated and can not be forgotten.

I would also wish to thank several staff members of the Computing Laboratory, in particular Dr. Isi Mitrani, for their many useful comments on this research. I would also like to thank Jonathan Spencer for his reading and commenting on parts of this thesis.

Finally, I would like to thank my sisters and brothers in Korea for the help and encouragement they have given me during the time I have spent working on this thesis.

Financial support for my scholarship during my studies was provided by grants from the British Council. Their support is greatly appreciated.

Table of Contents

List of Figures

List of Tables

Trademarks

UNIX™ is a trademark of AT&T in USA and other countries.

GNU™ is a trademark of Free Software Foundation in USA and other countries.

Sun Workstation® is a registered trademark of Sun Microsystems, Inc.

Smalltalk-80™ is a trademark of ParkPlace Systems.

Chapter 1
Introduction

Over the last two decades advances in semiconductor technology have allowed the development of small, low cost, powerful microprocessors and massive main memory computer systems. Changing technology has also greatly affected the design of future communications mechanisms. These new technologies suggest that basic assumptions that have held in the past may no longer hold in the future. For example, massive memory systems or randomly accessible secondary memory in local and remote systems make current memory access strategies significantly different from older methodologies. By combining the feasible hardware with advanced interconnection media, distributed systems are fast breaking out of their bounds and proceeding into larger technical and scientific communities. Besides the increase in computing power, flexibility can be achieved by using a distributed system: extra nodes may be added to the network as the demands on the system increase. Distributed systems also provide users with parallelism, load balancing and sharing, better utilization of specialized hardware, exploitation of resource locality, easier user mobility, fault tolerance and system maintenance.

In the distributed network environment, many processors interact to perform roughly equivalent tasks at less cost than in a centralized single processor computing environment. Remote execution[Popek 85, Bergland 86, Caceres 84, Ezzat 86] and process migration[Popek 85, Zayas 87, Finkel 86, Barak 85, Theimer 85] are typical computational methods used to achieve this goal. The execution of an operation on a resource in the network is also a computation that in turn may consist of a series of further operations on remote resources. In a real environment, the processor overhead of using any distributed system becomes less significant as the performance of the system is dominated by network latency. This is because microprocessors get faster but network latency remains roughly constant in spite of the development of new high-throughput networks. The performance of a distributed system can therefore be improved by preventing unnecessary network communication rather than by cutting the cost of basic network operations[Chase 90].

Computers today are configured with larger memories, but contention for memory usage remains an issue, especially on time-sharing systems. Users add more jobs to a machine until it runs out of some resources; often this bottleneck is memory [Breecher 89]. So, systems from personal computers to supercomputers with advanced memory management systems still require larger logical memory space than physical memory, as well as protection and sharing. This complex memory management is termed a virtual memory system.

A virtual memory system is vital to a distributed system because several of its nodes can support a global, uniform address space for network wide computations. This requires moving process images from one node to the other. When a virtual memory system performs memory management on local and remote systems, it is called a *distributed virtual memory system*. In these multiprocessor and distributed computing systems, the memory accessing policies become complicated and involve communication management. They therefore need different memory management strategies from conventional virtual memory systems. A variety of memory schemes are described in section 1.2.

1.1 Object Oriented Systems

Over the last few years there has been an increasing interest in object oriented programming languages and systems. Central to object oriented languages are facilities for data hiding, protection, extensibility and code sharing through inheritance and flexibility by the runtime binding of operations to objects. Object oriented languages provide flexible and efficient facilities for user defined types. A program can be partitioned into a set of objects that closely match the concepts of the real world problem. This technique for program construction is often called data abstraction. In an abstract data type, the details of object implementation are hidden – data hiding. So, the state of an object can only be accessed or changed by invoking one of the operations which form the public interface to the object. A class is defined in terms of a user–defined abstract data type. An instance of such a class is called an object. Objects of a user defined type contain information which consists of a data structure and a set of operations. Such an object encapsulates the data and operations of the object and defines an interface to other objects. A user program or client can create one or more instances of a class. An object consists of some data or member variables and a set of operations or member functions that can be applied to the object's data. Both the member variables and functions are defined by the object's class. A program cannot read or modify the member variables of an object directly because they are wrapped up by an interface. If the only way to modify or access the state contained within an instance of a class is to invoke one of the public operations, public member functions, then the data abstraction provided by the language supports encapsulation. When a new class is defined in terms of (and uses) the data and operations of an existing class, this is called inheritance. Programs that make use of encapsulation, inheritance and dynamic binding are called object oriented. The use of objects to structure programs enables modularity and software reuse[Stroustrup 86, Goldberg 83, Dixon 88, Parrington 88].

The memory management mechanism for the implementation of a pure object oriented architecture is different from that for imperative language systems. Two factors in object–oriented memory design greatly differ from conventional systems – the size of objects and the number of objects. For instance, Smalltalk objects are too small and too numerous to be managed individually[Kaehler 83, Kaehler 86, Kaiser 88]. Moreover, the representation of information in an object–oriented memory is completely different to that in conventional systems. Memory management must provide an interface at which logical entities, the objects, are presented while the actual hardware structure of the physical memory remains hidden. Also, objects are referenced by logical names rather than by physical addresses. An additional feature is the protection of objects from arbitrary accesses by introducing an access control mechanism. This feature is especially important when multiple processes share objects. Kaehler's LOOM, Kaiser's MUTABOR, Clouds[Dasgupta 90] and the Guide[Balter 90] system have implemented pure object handling storage management in local or distributed systems. As stated earlier, because of dealing with many small objects, I/O efficiency is rather low in these schemes, therefore, dedicated special hardware system may be required. Some systems[Campbell 88, Shapiro 89] implement their system software and application packages in object oriented languages. For performance reasons, these systems implement page based virtual memory in central or distributed systems rather than pure object managing storage management.

1.2 Requirements of a Virtual Memory in Central and Distributed Systems

Kaiser[Kaiser 88] has defined virtual memory management as having two major aspects, address space management and storage management. *Address space management* means the separation of logical address space from their physical address and the management of logical storage objects independent of their physical resource allocation. It includes the issues of protection and sharing of objects. The goal is to achieve convenient and secure object sharing between processes. For example, the start address of a UNIX user process is fixed irrespective of its physical image location in main memory. Also, some UNIX executable files keep process text and data in separate regions of running processes to permit protection and sharing.

Storage management deals with the allocation and deallocation of physical memory space. From the storage management point of view, a virtual memory conceptually separates logical memory from physical memory enabling the logical memory to be larger than physical memory. There are three principle methods for storage management; segmentation, individual object management and paging. In segmented memory schemes, information is structured into individual modules, termed memory segments. Each of these has a linear, contiguous, variable length of address space and contains data that logically belong together. One of the advantages of segmented memory schemes is their performance. It is more efficient to load a program all at once than to load it in small sections on demand. However, pure segmentation causes some serious difficulties for storage management. One drawback is that a segment must be entirely in main memory when in use. Although only small objects are referenced, an entire process image would be swapped in or out of memory.

Another challenge for pure segmentation is the handling of many small but persistent objects[Kaehler 86, Kaiser 88, Harland 87, Dasgupta 90] which is discribed in Section 1.1. This situation is an ideal candidate for object management in terms of memory utilization, protection, sharing and even object migration. Moreover, a new approach taken in object oriented virtual memory management is that objects (usually, persistent objects) are units of fetching and purging in local secondary memory management as well as in the distributed shared memory system. Clouds[Dasgupta 90] even implemented a concept of persistent objects in a page based virtual memory management. However, because every object is independently swapped between a real memory and a disk or other storage medium, input and output efficiency may be rather low. Also, a number of index tables are required to manage the small individual objects either stored in a main memory or on a disk. So some systems[Harland 87] combined the pure object fetching into a conventional paging system with hardware support in order to compensate for the performance degradation of pure object management. A notable point in managing individual objects in virtual memory systems is that object fetching mostly relys on on-demand fetching in spite of the large number but small size of objects.

For efficient storage management, we find demand paging to be the most commonly accepted strategy[Leffler 89, Kaiser 88, Hwang 84] in conventional computing systems. Paged memory offers a single linear address space. As a program runs, additional sections of its program and data spaces are paged in on demand. However, paging is not a panacea for object-oriented memory management. The fixed size of pages does not meet the individual requirements of objects because a unit of page is dependent upon the hardware rather than the granularity of object oriented programs. As a testbed to implement object-oriented accurate prefetching(AP) in the future, a page-oriented virtual memory system is adopted to achieve a simple and efficient implementation of the prefetching using an existing program execution environment. The algorithm to be developed in this thesis can be applied for conventional paging systems as well. Moreover, in terms of address space management, pages offer poor support for individual object protection and sharing because a page may consist of many small objects. However, a paging system is discussed as a storage management system in this thesis because they have several advantages[Leffler 89]:

- *Allows large programs to run on small memory configurations. It allows more programs to be resident in main memory to compete for CPU time, as the programs need not be completely resident. When programs use sections of their program or data space for some time, leaving other sections unused, the unused sections need not be present.*
- *Allows programs to start up faster, as they generally require only a small section to be loaded before they begin processing arguments and determining what actions to take. Other parts of a program may not be needed at all during individual runs.*

Each virtual address system in a distributed system has a larger address space than any single physical address space can provide in a single node. This is because the distributed address space can be shared by all nodes in a distributed system. Young gained some insight into the proper role of location independence by looking at a network wide virtual memory system combined with communication[Young 86, Dafni 87]. Some workstations that utilize remote file servers, combined with larger main memories, make it worthwhile to increase the amount of prepaging[Leffler 89, Duglis 87].

In Li's shared virtual memory for multiprocessor systems[Li 89, Scheurich 89] and other distributed systems[Li 86, Fleisch 87], address space is organized in pages which can be accessed by any node in the system. A paragraph from Li's recent paper[Li 89] points out that:

> *The shared virtual memory not only "pages" data between physical memories and disks, as in a conventional virtual memory system, but it also "pages" data between the physical memories of the individual processors. Thus data can naturally migrate between processors on demand. Furthermore, just as a conventional virtual memory swaps processes, so does the shared virtual memory. Thus the shared virtual memory provides a natural and efficient form of* process migration *between processors in a distributed system. This is quite a gain because process migration is usually very difficult to implement. In effect, process migration subsumes remote procedure calls.*

In particular, Fleisch's model[Fleish 87] assumed that a fast (up to 8 Gigabits/second), high bandwidth network is available. Disk transfer rates will also increase, but probably less dramatically than for buses and networks. These increased rates suggest distributed shared memory will become more attractive in the future because large memory systems will continue to be limited more by their I/O capacity than by memory. A shared memory facility could reduce the number of I/O operations and therefore improve system performance. Fleisch assumed that distributed shared memory design should be similar to a virtual memory paging system. A memory mapping manager on each node views its local memory as a large cache of pages for its associated processor. A memory reference causes a page fault whenever the page containing the memory location is not in a processor's current physical memory. When the fault occurs, the memory mapping manager retrieves the page from the memory of another processor, so called memory–to–memory access. If there is a page frame available on the receiving node, the page is simply moved between the nodes. Otherwise, the shared virtual memory system uses page replacement policies to find an available page frame, swapping its contents to the sending node. This paging mechanism is similar to that on a diskless workstation.

When disks are present in the shared or distributed virtual memory environment, they can easily be incorporated into the memory hierarchy of the system. If a disk server provides a transparent paging service then the client disks can be seen as a remote memory. However, the way of managing page faults in shared or distributed memory access is different from that for conventional disk based page faults. Remote memory can serve as an added level of the memory hierarchy between local memory and disks[Li 89]. In this way, the full potential of the large aggregate memory of a multicomputer can be utilized. Application programs are completely relieved from having to arrange data movement and availability across the network in an effort effectively to use the capabilities of a multicomputer. However, in managing page faults, remote memory can allow random accessing of any pages in the memory which, in contrast, is very expensive in disk management. Also, the cost of communication is an important factor compared against disk seek time[Cheriton 88, Theimer 85].

The major difficulty, though, is the cost of transferring a computation's context from one system node to another. This context which consists primarily of the process virtual address space is typically large in proportion to the usable bandwidth of the interconnection medium. Moving the contents of a large virtual address space thus stands out as the bottleneck in remote memory access and process migration. As programs continue to grow, the cost of migrating them by direct copy will also grow linearly. Any attempt to make process migration a more usable and attractive facility in the face of large address spaces must focus on this basic bottleneck. One approach is to perform a logical transfer, which in reality requires only portions of the address space to be physically transmitted. Instead of shipping the entire contents at migration time only part of the referenced page can be sent. If on a page based process migration system, during a program execution, the text file is not stored at the execution site the page could be demand copied across the network. Zayas and Popek[Zayas 87, Popek 84] showed that efficient process migration could be achieved by paging portions of a migrated process only as they were needed. It makes it possible to start executing a migrated task before moving all of its pages onto the new host. Pages can be faulted across the network and moved by copy–on–reference or demand prefetched.

Shared memory multiprocessor systems have another problem associated with memory coherence in their cache memory. A program running on a multiprocessor no longer has a single, sequential order of execution. The locality of reference of a processor is easily disturbed by the actions of other processors. Parallel computing introduces a new type of problem in multiple cache memory systems. In conventional uniprocessor systems, the higher the locality of data, the better the system performance that can be achieved. However, this high locality of reference for a block of data may cause problems when different processors modify adjacent locations. For example, the first write transfers the block to one processor's cache. The second write moves it to another processor's cache. This sequence is called *false sharing*[Hill 90] since no information is transferred. False sharing arises when the data of two processors lie adjacent in memory in the same page. So, reducing cache misses is more complex on a multiprocessor due to interactions with other processors and it is quite different from that in uniprocessors. For example, a cache memory system that keeps more items in the cache by packing them tightly may introduce false sharing between processors, thus, degrading performance. Programmers should not optimize multiprocessor programs for finite caches unless the amount of data each processor uses is large and the changes do not cause harmful interactions with other processors. Therefore, a new scheme is required to manage this kind of cache miss by adopting a new strategy for caches in multiprocessor systems.

1.3 Related Mechanisms

The concept of virtual memory was originally proposed in ATLAS[Edwards 64] and evolved in the Multics[Organic 72] project. Research on traditional virtual memory management for uniprocesor architectures[Denning 79] has had significant impact on developing modern high performance computer systems. Locality of reference was the most important observation about sequential programs. The use of virtual memory, however, can degrade performance. There is a finite cost for each operation, including saving and restarting state and determining which pages must be loaded[Leffler 89]. Techniques such as grouping, restructuring and prefetching have been developed to augment the performance of a virtual memory system.

1.3.1 Grouping and Restructuring

The modern program design that encourages small single purpose modules like objects could incur a side effect in the execution of the programs. If these modules are linked together in a random fashion in terms of control flow, as is commonly done, then an operation often requires access to many modules scattered over numerous pages of memory. Such a practice clearly leads to a poor locality of reference. Unfortunately, program reference patterns can be extremely complex: because of this, programmers generally find it difficult to determine the locality of their code either by observation or through the use of tools. There are techniques for improving the locality of code reference by grouping or restructuring modules at compile time or runtime. These methods involve automatic or semi-automatic reorganization of the code (in essence a rearrangement of the link line) to optimize the location of modules[Breecher 89]. These techniques can be applied to some languages having poor locality of reference. Functional languages like Lisp or dynamic binding based object oriented languages like Smalltalk[Courts 88] are good examples of low locality of reference. Stamos and Williams describe static and dynamic grouping in the Smalltalk environment[Stamos 85, Williams 87]. More details of these works will be given in Chapter 2. However, most of the work is language specific and there is no general solution by this approach.

1.3.2 One Block Lookahead (OBL)

A typical prefetching technique is based on spatial locality and temporal locality. Such an example of a prefetching algorithm exploiting these properties is the one block lookahead (OBL) algorithm. Horspool described Joseph's original work on OBL in his paper[Horspool 87];

> *In early 1970, Joseph simulated the policy. Thus, if a faulting reference to page number* i *occurs and the page numbered* i + 1 *is not resident in main memory either, then both pages* i *and* i + 1 *are loaded into main memory. This is beneficial because it should not require twice as much work for the operating system to fetch two consecutively numbered pages together as it would to fetch just one of the pages. Most secondary memories are disk based and there would be little extra seek time and latency delay needed to read the second page.*

In spite of OBL's inaccurate prefetching, it has been widely used in many operating systems because of its simplicity and potentially minimal disk seek and latency time based on spatial locality. However, the mis-prefetchings may break a working set, in particular in local replacement algorithms. Moreover, it cannot be used for a lookahead non-consecutive page because it is unable to encompass branches or long distance calling in a program. Furthermore, suppose OBL is used on a distributed memory access or in a diskless workstation, it is far from an optimal methodology since the effect of mis–prefetching is more expensive than on a local disk based system. Therefore, an optimal prefetching policy which will work well on a distributed memory access is required in local as well as distributed systems. Also, we need a new prefetching policy to lessen mis–prefetching and causing less resident set breaking than OBL.

1.4 A Prefetching System Model

The model assumed in this thesis is a distributed computing system which supports prefetching of objects or pages in disks or networks. As shown in Figure 1.1, any node in a distributed computing system can be either a client or a server depending on the processes running on the machines. This model can include a dedicated file server model, for example the Bullet[Renesse 89] file server which has a large RAM memory for caching disks. A major point in this model is that the CPU overhead of distributed operations becomes less significant as the performance of the system is dominated by network latency, and will remain so despite the advent of new high–throughput networks. The performance of a distributed system depends on the degree to which the system prevents unnecessary network communication [Chase 89]. So object or page migration based on prefetching is an alternative of on–demand paging in distributed systems.

There are two major prefetching operations that may be carried out in the model. The first is a conventional virtual memory function which operates between a main memory and a disk based secondary memory system. The data transfer medium between a main memory and a disk is a bus. The second is prefetching based objects or page migration between nodes. Here, the communication medum is assumed to be a local area network which has non–negligible communication overhead compared to disk access times. In addition, one more prefetching operation in the model can be assumed between very high speed CPU cache memory and main memory but this is not considered in any detail here. Because cache memory is based on a line fetching which is of relatively small

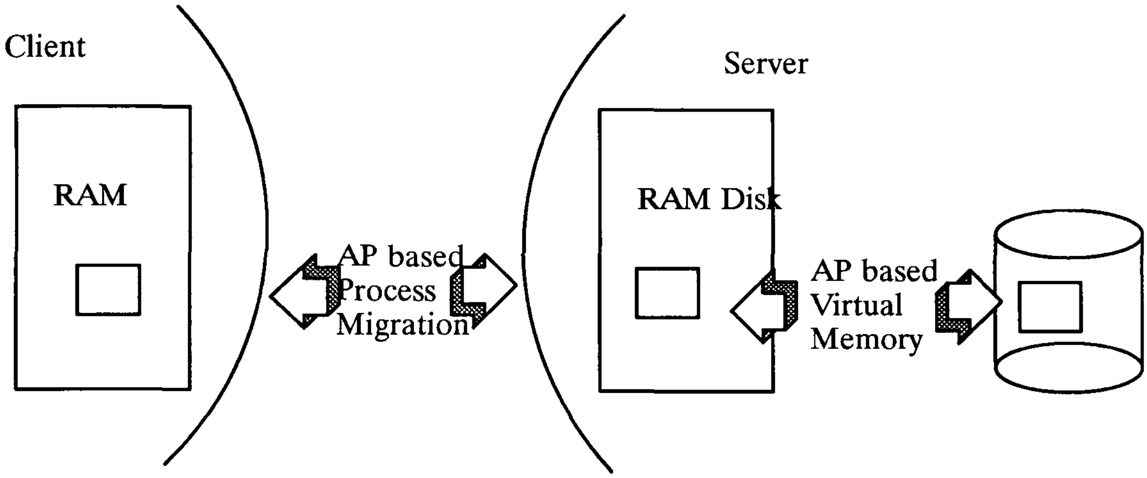


Fig.1.1 The Model of a distributed computing system

size compared to a page and it requires special hardware such as associative memory for high speed lookup of reference tables. The lookup time should be within the order of several machine cycles.

Paging is adopted througout the implementation of accurate prefetching(AP) in this thesis because the principle of accurate prefetching of objects or pages which contain objects are the same. Also, when the paging system is adopted in the simulation, it enables to implement a quick prototype to demonstrate the feasibility of AP but other issues like memory utilization, object indexing and management are considered as outside the scope of this thesis.

1.4 The Aims of this Thesis

The basic concept of prefetching in virtual memory systems, to run paging system where some fetches of pages into the main memory were performed before any reference to those pages had occurred, was known in early 1970 [Horspool 87]. Studies of prefetching process images have been reported by Joseph[Jorseph 70] (one block lookahead), Baier[Baier 76] (whose PRED function is based on spatial lookahead and recurrent patterns), Horspool (whose PRIO function resolved the memory inclusion problem) and Theimer's preswapping paradigm[Theimer 85]. These predictive studies present complicated priority techniques or statistical analysis and could reduce memory pollution by suppressing unnecessary prefetches of pages in one block lookahead but still could not encompass more than OBL. The term *memory pollution* means the phenomenon of wasting main memory with useless prefetched pages which are not referenced within a reasonable period of time[Horspool 87].

As object oriented programming is becoming used widely in software development, the requirement for an advanced virtual memory system scheme for object management in new hardware systems is growing. As stated earlier, there are no accurate prefetching algorithms applicable to an object oriented distributed shared memory system or diskless workstations. The nearest example is Zayas'[Zayas 87] copy–on–reference scheme in which a page migration is delayed until there is a reference to the page. When copy–on–reference was realized on a network a problem was posed. It is that there are as many remote page requests in the distributed system as page faults in local execution and this creates heavy loading for the network service routines and increased communication overhead. Also, the copy–on–reference did not show good results in migrating non–Lisp programs. Zayas pointed out that if there were an optimized prefetching algorithm available to reduce the distributed page faults the bottleneck of a process migration and distributed memory access would be resolved dramatically.

A working set model based on locality of reference which will be discussed in detail in Chapter 2. In this model, many page faults are incurred at the phase transition of working sets. This is shown in the Figure 1.2 where the x axis shows time progressing and the y axis is the number of page faults [Williams 87]. The initial paging stage is unavoidable in a pure paging system but the time can be reduced by adopting a

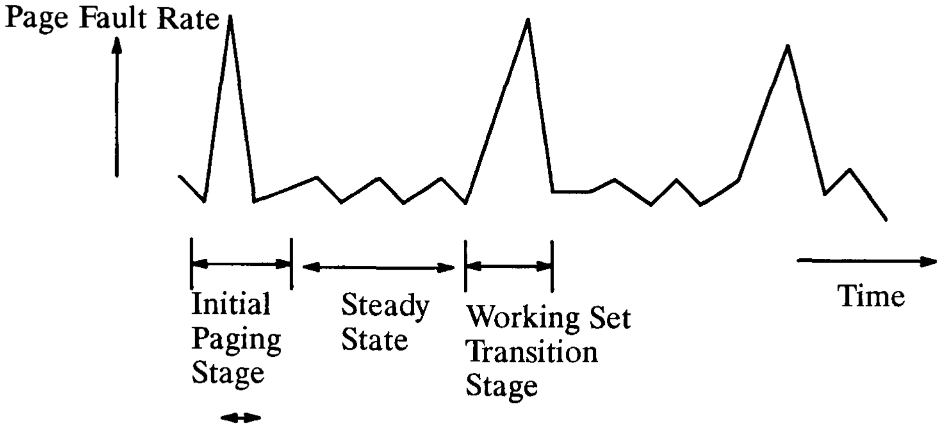


Figure 1.2 A working set transition diagram

segmented page scheme. After loading the first working set into main memory, the page fault rate is decreased sharply for a time being until a transition phase follows. This is because the working set satisfies a reference string of a processor. However, in a virtual memory system this working set changes into another working set through a number of page faults at the working set transition stage. As this figure shows, there is a high rate of page faults at the initial paging stage and at every working set transition stage. These high paging rates cannot be managed by a natural paging scheme or a simple prefetching scheme but an accurate prefetching scheme can resolve it.

Moreover, the way of managing object or page faults in a shared or distributed memory access is different from that in disk based faults. Most conventional virtual memory systems work on just disk based systems. However, network expansion of virtual memory systems requires an advanced prefetching policy to support efficient object or page movement in memory–to–memory or memory–to–remote disk server in distributed systems. This is because remote memory allows access to random objects or pages as cheaply as contiguous object or page access and, compared to communication overhead, disk seek time is no more the dominant factor[Cheriton 88, Fleisch 88, Johnston 90]. This enables the prefetching a group of objects or pages independent of their physical location in remote memory. Most efforts to date have centered on increasing the fetching system performance by considering mechanical disk head movement. But, a new mechanism to deal with the faults in terms of random accessing secondary memory should be provided in a distributed virtual memory system.

Furthermore, because of the modular subdivision of a program (in an object–oriented languages) into scattered objects and the separation of code from data in central and distributed environments, the simple sequential prefetching policy that has been performed in OBL is insufficient for high performance computing systems. As in normal virtual memory, prediction of future accesses in OBL is based on past history[Brent 87]. As this is invalid, in particular, there is no high serializability in data pages. Many exceptions are generated by object invocations during the transition to a new phase. So, OBL is not very accurate and cannot suppress the fault rates caused by object invocations. This thesis describes how to construct an accurate demand object page prefetching(AP) with low memory pollution for object oriented systems. Object oriented programming languages have good properties for building inter object relationships using inheritance as well as establishing an intra objects relationship between separated object member functions and their variables through encapsulation. An interclass dependency which may cause the execution of methods up and down the class hierarchy, the so called *yoyo* phenomenon[Taenzer 89], is used for AP. Also, overprefetching is controlled by combining the inter object relationships with branch based control flow analysis method.

Control flow graphs have been used for virtual memory by Verhoff[Ver 71], Hatfield[Hatfield 71], Stamos[Stamos 84], Brent[Brent 87] and Hartley[Hartley 88]. These studies presented several grouping[Stamo 84, Williams 87] or restructuring[Verhoff 71, Hatfield 71, Breecher 89, Hartley 88] methods as well as prefetching[Brent 87]. Although the restructuring improvements may only be beneficial for the particular traces that were studied, a different set of input data even for the same program can alter the trace sufficiently so that the restructured program runs worse than the original version[Brent 87]. Brent reported an accurate prefetching policy using program structure and a special hardware unit for cache memory line prefetching. However, the prefetching is unable to demonstrate good results using control flow analysis because the special hardware is unable to trace all CPU control flow at every branch point. More details of Brent's work are given in Chapter 2. Also, Kaehler[Kaehler 83] investigated an object based swapping (instead of page or segment swapping to save main memory) but showed that the relative cost of object swapping versus paging is high. This is explained in greater detail in section 2.4. However, the use of object oriented programming structure and control flow graph for a virtual memory system has neither been suggested nor studied in detail. The accurate demand prefetching policy (AP) described in this thesis could supress all such exceptions that may occur in long jump by making use of control flow analysis.

In addition, AP implies a logical grouping for object pages in a virtual memory system. This is because AP is obtained by module connectivity and entities are dispersed over many pages. Also, AP can further augment performance when it runs on top of some physical groupings such as in a Smalltalk environment. A static physical grouping is partly required in AP for some dynamic objects to combine together. On the other hand, some shortcomings posed by the physical grouping can be reduced by incorporating an accurate logical grouping. The logical grouping does not move or duplicate any modules physically but it does prefetching of objects or pages which it is possible to group and read together into main memory.

In a reliable distributed system, sending large packets of data is cheaper than sending several small ones[Li 89b]. This fact provides the justification for sending groups of objects or pages. Consequently, AP consumes less overhead in terms of faulting rate and processing requirements. Eventually, AP will be used as a transparent integrated virtual memory function for local operation (to and from disk) and a distributed operation (to and from remote machines).

1.5 Thesis Structure

This thesis is structured as follows. Chapter two expands on general background in terms of the virtual memory system model, locality of reference, replacement algorithm, restructuring, grouping and prefetching. The chapter then discusses and compares virtual object memory management policies in a number of projects with goals similar to those of AP.

Chapter three begins by formulating the AP system with regard to reference strings and disk queue sorting. It then considers in greater detail the properties influencial to object prefetching in object oriented languages. In the following two sections discuss locality of reference and the yoyo problem in object-oriented programming languages in order to establish prefetchability. This chapter also examines object data prefetching in detail.

Chapter four deals with the conceptual basis of control flow graphs and building prefetching blocks at compile time. This description of control flow graphs is followed by a discussion of symbols to represent the control flows for a given program. Then the use of prefetch blocks in a paging system is described in detail. Some considerations of control flow in early and late binding object-oriented languages are described in the following section.

In Chapter five it is shown how the mechanisms described in Chapters three and four can be implemented to demonstrate the feasibility of AP. The first section of the chapter describes in detail the implementation of the virtual memory simulator. The procedures for constructing the compiler generating the prefetch tree are described in the following sections. The remainder of the chapter describes the employed mechanisms by the simulator in using the prefetch table.

Chapter six shows performance measurements of the experimental prototype of the compiler and the simulator. The chapter begins by illustrating performance measurements for the compiler and then the results are compared to an original compiler. General methodologies of performance analysis for prefetching schemes are described and the influence of how fault rate on the virtual memory system is discussed. The following section illustrates measured performance results in detail. The final section analyzed the results and discusses various points affecting the performance and influences of AP on a whole computer system.

The final chapter provides a summary of the thesis, presents the conclusion of this thesis, and suggests some future research.

Chapter 2
Virtual Memory Issues and Prefetching

The previous chapter described how virtual memory systems are unable to provide efficient support for the latest object oriented virtual memory management schemes in local and distributed systems. A new accurate prefetching scheme is desirable so that it can prefetch a group of distributed objects or pages thus reducing the number of page faults. Also, it should be able to support new types of secondary memory (for example, RAM disk[McKusick 90]) and process migration. In order to build such an accurate prefetching method, we need an understanding of the details of a virtual memory system model, program locality, paging schemes, object oriented system model, various types of grouping and restructuring and prefetching. This chapter expands on these virtual memory issues and discusses the requirements of accurate prefetching.

In the first section of this chapter, the description of the virtual memory system model gives an overview of how a hierarchical memory model can be built and its operations, organization and other related issues are described. This section also introduces the basic techniques used to create a large virtual address space and the mechanisms necessary for translation to physical space. To explain memory reference patterns in a computing model, two types of program localities are considered in the following section. Section 2.3 discusses some hardware requirements for the implementation of virtual memory systems. Section 2.4 describes the details of how paging systems operate using page fetching and replacement algorithms in fixed and variable spaces. Various policies of memory replacement are presented in this section. Section 2.5 considers various techniques for program restructuring. Techniques for augmenting the performance of virtual memory in object-oriented systems are summarized in the three following subsections. Section 2.7 describes various prefetching policies in greater detail by reviewing the policies whose aims are similar to the work of this thesis.

2.1 Virtual Memory System Model and Terminology

Hierarchical memory systems provide a huge virtual address space with low cost hardware. This section describes the memory model, taxonomy and how the system can be operated.

2.1.1 Memories

The virtual memory model and terminology in this section are based on those of Hwang[Hwang 84], Leffler[Leffer 88] and Silberschatz[Silberschatz 88]. Memory systems for conventional computers are hierarchical memory structures. The design objective of hierarchical memory is to attempt to match the processor speed with the rate of information transfer or the bandwidth of the memory at the lowest level and at a reasonable cost. Memories in a hierarchy can be classified in terms of access method and speed or access time.

Firstly, there are three kinds of access method available – *random–access memory (RAM)*, *sequential–access memory (SAM)*, and *direct–access storage devices (DASDs).* In RAM, the access time of a memory word is independent of its location. In SAMs, information is accessed serially or sequentially (for example magnetic tapes). DASDs are rotational devices made of magnetic materials where any block of information can be accessed directly. The moveable arm disk is the most common DASD. The disk requires "seek time" to move the arm to the desired track.

Secondly, memory can be classified into primary memory and secondary memory in terms of the speed or access time. Primary memory is made of RAM and it is termed a main memory. Secondary memories are made of DASDs and optional SAMs.

In some workstation environments[Cheriton 88, Holliday 88], the common two-level hierarchy is becoming a three-level hierarchy, with the addition of file-server machines connected to a workstation via a local-area network. Also, distributed memories require a wide variety of memories distributed throughout the system. In the Cedar[Swinehart 86] system, for example, each processor has its own local memory, each cluster of processors has a cluster memory, and the entire system has a global memory which is accessible by all processors.

2.1.2 Virtual Memory System

A program consisting of objects is translated to modules of machine code and unique identifiers by a compiler. A linker then combines these modules of unique identifiers and a loader translates the unique identifiers into main memory locations. The set of the identifiers defines the *virtual space* or the *name space* and the set of main memory locations allocated to the program defines the *physical memory space*. If a system provides a mechanism for translating the program generated virtual addresses into the memory location addresses, this is called *virtual memory system*. Usually, virtual memory allows a larger virtual space than physical memory space. A *virtual machine* is defined by the architecture of the hardware on which a process executes. References to a virtual address space are translated by hardware into references to physical memory. This operation, termed *address translation*, permits programs to be loaded into physical memory at any location without requiring position-independent addressing.

When an item is referenced at time t, it might be that the item is not be located in physical memory. This is called an *addressing exception* or *missing item fault*. When a missing fault occurrs, a fault handler brings in the required item from the next lower level of memory. Because the fault rate is so significant in modeling the performance of a hierarchical memory, memory management policy is often characterized by a probability for finding the requested information in the memory of a given level. This is called the hit ratio H. Several factors effect the hit ratio, such as memory size, granularity of data transfer, memory management policies. Suppose a hit function is H(s), the miss ratio must therefore be F(s) = 1 - H(s).

In a generalized memory hierarchy, the missing item is retrieved by requesting the item at successive lower levels. Three basic policies, termed *fetch policies*, define the control of the transfer of the missing item from lower level to the desired level. A *fetch policy* decides when an item is to be fetched from lower level memory. A variation of fetching policies are described in detail at Sections 2.6 and 2.7. A *placement policy* selects a location in memory where the fetched item will be placed. Where the memory is full, a *replacement policy* chooses which item or items to remove in order to create space for the fetched item. Section 2.4.1 describes one in detail.

2.2 Program Locality

Programs tend to favour a relatively small portion of their images during their execution. Therefore, the virtual addresses generated by a process are not very random but behave in a some predictable manner. This characteristic is referred to as the *locality of reference* and describes the fact that over an interval of virtual time, the virtual addresses generated by a typical program tend to be restricted to small sets of its name space, as shown in Figure 2.1. The reference data of Figure 2.1 was observed from an execution of a C++ program on UNIX. Program looping, sequential execution and block structure construct several groups of locality of references in a program. This locality of reference is broken at the transition of control from one group to another as shown in Figure 2.1 Also, if one considers the interval Δt in Figure 2.1 the subset of pages referenced in that interval is less than the set of pages addressable[Baier 76, Hwang 84].

There are two kinds of locality of reference: *temporal* and *spatial*. Temporal locality is the tendency of a program to reference during the process time interval $(t, t + \Delta t)$ those pages which were referenced during the interval $(t - \Delta t, t)$ [Baier 76]. Thus, over

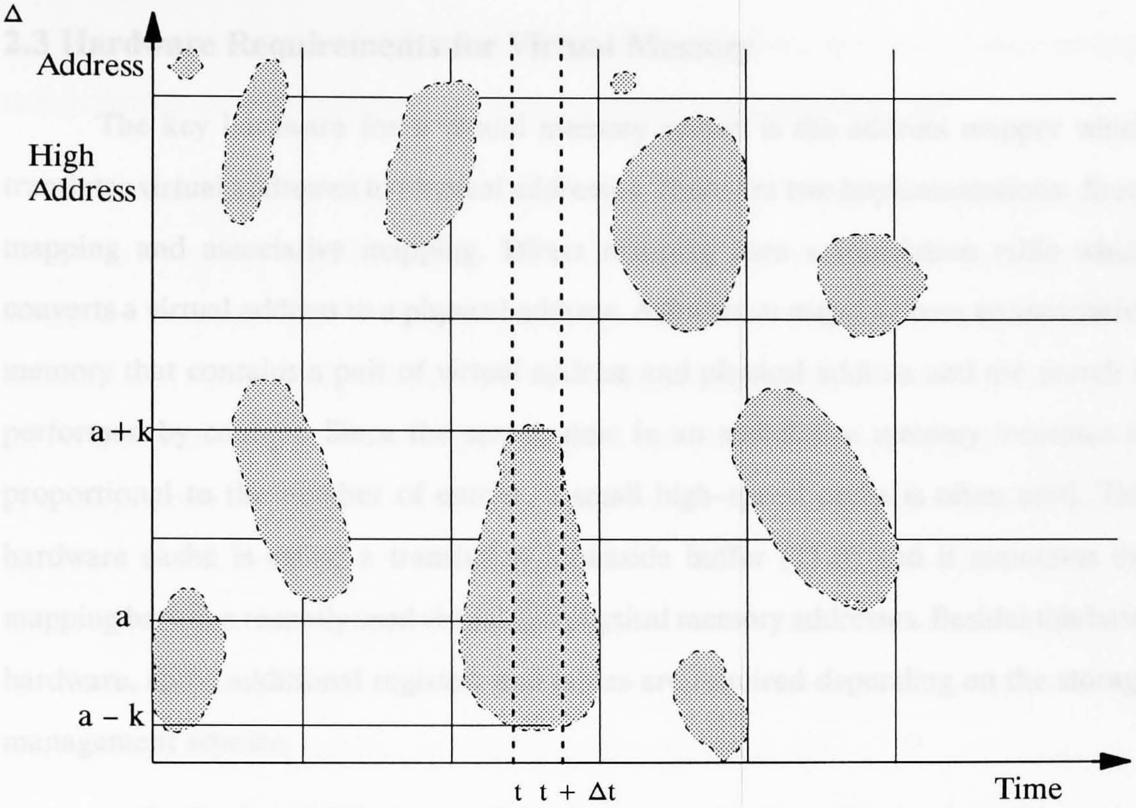


Fig.2.1. A storage reference pattern on UNIX

short periods of time a program references memory nonuniformly, but over a large period of time, portions of the address space which are favored remain largely the same. This behavior has been observed in program loops, the use of temporary variables and stacks of processes. Spatial locality means that if a process is to make references to an address space in the future it is likely to be near the current location of reference. Namely, if *a* is the address referenced at time *t*, spatial locality of reference is the address space (*a* – *k, a* + *k*) during time (*t, t* + Δ*t)* in Figure 2.1. Naturally, temporal and spacial locality coexist during the execution of a program. Traversals of a sequential set of instructions and arrays of data enforce spatial localities. Notice that spatial locality is an important factor in deciding the size of the block to be fetched and temporal locality effects the determination of the number of blocks in a segment.

2.3 Hardware Requirements for Virtual Memory

The key hardware for a virtual memory system is the address mapper which translates virtual addresses to physical addresses. There are two implementations: direct mapping and associative mapping. Direct mapping uses a translation table which converts a virtual address to a physical address. Associative mapping uses an associative memory that contains a pair of virtual address and physical address and the search is performed by content. Since the search time in an associative memory increases in proportional to the number of entries, a small high–speed cache is often used. This hardware cache is called a translation lookaside buffer (TLB) and it maintains the mapping between recently used virtual and physical memory addresses. Besides this basic hardware, some additional registers and tables are required depending on the storage management scheme.

Leffler[Leffeler 89] discussed hardware requirements for implementing page based virtual memory as the following:

- Hardware support for the collection of information on program references to memory.
- The CPU must distinguish between resident and non–resident portions of the address space.
- When the system selects a page for replacement, it must save the contents of that page if they have been modified since the page was brought into memory.
- The hardware maintains a per–page flag showing whether the page has been modified.

Many machines also include a flag recording any access to a page for use by the replacement algorithm. Also, a special instruction set is necessary to support a paging system. If a processor stops execution during an instruction before it ends, the processor must be able to restart the instruction after handling the page fault because intermediate computations done before the page fault may have been lost [Bach 86].

To handle many very small objects in virtual object memory systems, special hardware units have been developed. For example, because each object is independently swapped between real memory and the disk, MUTABOR[Kaiser 88] and REKURSIV[Harland 87] adopted individual object fetching systems. Two level index tables and accessing methodology were fully implemented in hardware to be able to provide performance requirements. In the case of Brent's[Brent 87] prefetching, special hardware was used to trace CPU execution. This is described in Section 2.7.3. Also, the Ivy[Li 86, Li 89b] system (network wide shared virtual memory) maintains memory coherence by using virtual memory hardware to implement page ownership schemes analogous to hardware cache consistency protocols[Chase 89].

2.4 Paging

Virtual memory can be implemented in many ways, some of which are software based, such as overlays. The most effective virtual memory schemes are, however, hardware based. In these schemes, the virtual address space is divided into fixed sized units, termed pages. Virtual memory references are resolved by the address translation unit to a page in main memory and an offset within that page. Page security, i.e. privacy protection is applied by the hardware of the memory management unit on a per page basis.

Address translation handles the first requirement of virtual memory by decoupling the virtual address space of a process from the physical address space of the CPU. To satisfy the second requirement, each page of virtual memory is marked as *resident* or *nonresident* in main memory. If a process references a location in virtual memory that is not resident, a hardware trap termed a *page fault* is generated. The servicing of page faults permits processes to execute even if they are only partially resident in main memory.

In normal circumstances, all pages of main memory are equally good and the placement policy has little effect on the performance of a paging system. Thus, a paging system's behavior is mostly dependent on the fetch policy and the replacement policy. Under a *pure demand–paging* system, a demand–fetch policy is used, in which only the missing page is fetched and replacements occur only when main memory is full. In practice, however, few paging systems implement a pure demand–paging algorithm. Instead, the fetch policy is often altered to prepaging, which pages are fetched into the main memory before any reference to those pages occurs, and the replacement policy is invoked somewhat sooners than actually required. Hence, some in–use pages may be replaced with pages which may or may not be used.

2.4.1 Page Replacement Algorithms

Page replacement algorithms can be put into two major classes: fixed space replacement and variable space replacement. If main memory allocation is fixed for a user program the number of page fetching and purging in and out of main memory are necessarily matched. Thus, fixed space replacement algorithms can be implemented locally (on a per process basis) which chose a process for which to replace a page, and then chose a page in the process. The definitions of these algorithms are discussed in the following subsection. On the other hand, if main memory allocation is allowed to vary, fetching and replacement are able to happen independently in variable space replacement. The definitions of global replacement algorithms (one in which the choice of a page for replacement is made according to system wide criteria) are described in the following subsection.

2.4.1.1 Fixed Space Replacement Algorithms

Commonly used demand page replacement memory policies for fixed space attempt to take advantage of temporal locality to approximate the longest time to next reference replacement strategy [Hwang 84, Denning 79, Baier 76]. The behavior of the *i*th process is described in terms of its *reference string* which is a sequence:

R*i*(T) = r*i*(1) r*i*(2)...r*i*(T)

where r*i*(*t*) ∈ V*i* is the *t*th virtual address generated by process *i*. A reference string R is a sequence of T references r(1) ... r(t) and a *resident set* Z(t) is a subset of all the program's segments present in the main memory at a given time *t*. If the reference r(*t*) is not in the resident set established at time *t* – 1, a *page fault* occurs at time *t*. A useful measure of a process's behavior is the *fault rate*, which is the number of page faults encountered while processing a reference string, normalized by the length of the reference string. The most common method used in measuring the effectiveness of a page–replacement algorithm is the fault rate. So, the best choice of a page to replace is the one with the longest expected time until its next reference. The *forward distance* at time *t* for page *x* is the distance of the first reference to *x* after time *t*. This requires a *priori* knowledge of the paging characteristics of a process. Similarly, we define the *backward distance* as the distance to the most recent reference of *x* in R(*t*).

$\boxed{1}$ Least Recently Used (LRU) – At a page fault, it replaces the page in Z(*t*) of the process with the largest backward distance.

$\boxed{2}$ Belady's optimal algorithm (MIN) – At a page fault, it replaces the page in Z(*t*) with the largest forward distance. This algorithm minimizes the number of page faults but it is not practical.

$\boxed{3}$ Least Frequently Used (LFU) – Replaces the page in Z(*t*) that has been referenced the least number of times.

$\boxed{4}$ First in First out(FIFO) – Replaces the page in Z(t) that has been in memory for the longest time.

$\boxed{5}$ Clock algorithm (Clock) – This is a combination of a FIFO queue which is made circular and the establishment of a pointer for the circular queue and usage bit in each queue entry. The usage bit for an entry in the queue is set upon initial reference. On a page fault, the pointer resumes a cyclic scan through the entries of the circular queue, skipping used page frames and resetting their usage bits. The page frame in the first unused entry is selected for replacement. This algorithm attempts to approximate LRU within the simple implementation of FIFO.

$\boxed{6}$ Last in First out (FIFO) – Replaces the page in Z(t) that has been in memory for the shortest time.

$\boxed{7}$ Random (RAND) – Chooses a page in Z(t) at random for replacement.

2.4.1.2 Variable Space Replacement Algorithms

Variable space page replacement algorithms are an extension of commonly used fixed space replacement policies. One approach is simply to apply the replacement rule to the entire contents of main memory without identifying which process is using a given page. Examples of this approach are as follows [Hwang 84, Denning 79, Baier 76]:

$\boxed{1}$ Global LRU arranges all the pages of the active processes into a single global FIFO stack. Whenever an active process runs, it will reference its locality set pages and move them to the top of the global LRU stack. At a page fault, it replaces the page with the largest backward distance in the system.

$\boxed{2}$ Global FIFO arranges all the pages of the active programs into a single global FIFO list. It replaces the page that has been referenced the least number of times in the system.

[3] FINUFO – In global FINUFO (first in, not used, first out), all the pages of the active processes are linked in a circular list with a pointer designating the current position. Each page has a usage bit which is set by the hardware when the page is referenced. Whenever a page fault occurs, the memory advances the current position pointer around the list clearing set usage bits and stopping at the first page whose usage bit has already been cleared. This page is selected for replacement. This paging algorithm was used in the Multics system.

[4] Working Set(WS), W(t, θ), is used to denote an estimator of a locality set. W(t, θ) of a process at time t is defined as the set of distinct pages which are referenced during the execution of the process over the interval (t – θ, t) where θ is the window size. The working set size w(t, θ) is the number of pages of the set W(t, θ). This algorithm retains in memory exactly those pages of each process that have been referenced in the preceding θ seconds of process (virtual) time. If an insufficient number of pages are available then a process is deactivated in order to provide additional page frames. Notice that the working set policy is very similar to the LRU policy in that the working set algorithm specifies the removal of the LRU page when that page has not been used for the preceding θ time units whereas the LRU algorithm specifies the removal of the least recently used page when a page fault occurs in a memory capacity. The success of the working set algorithm is based on the observed fact that a process executes in a succession of localities: that is, for some period of time, the process uses only a subset of its pages and with this set of pages in memory, the program will execute efficiently. This is because, at various times, the number of pages used in the preceding θ seconds (for some appropriate θ) is considered to be a better predictor than simply the set of K (for some K) pages most recently used [Denning 79, Hwang 84, Bach 86].

However, no current computer system uses the true working set policy for paging or true LRU for the cache memory because of the expensive hardware support required for an implementation[Horspool 87]. The 4.3 BSD virtual memory system[Leffler 89] does not use the working–set model because it lacks accurate information about the reference pattern of a process. It does track the number of pages held by a process (the resident–set size), but it does not know which of the resident pages constitute the working set: the count of resident pages is used only in making decisions on whether there is sufficient memory for a process to be swapped in when that process wants to run.

5 VMIN – This is an ideal variable space memory policy which could be local or global. VMIN generates the least possible fault rate for each value of mean resident set size. At each reference $r(t) = i$, VMIN looks ahead: if the next reference to page i occurs in the interval $(t, t + \theta)$, VMIN keeps i in the resident set until that reference otherwise VMIN removes i immediately after the current reference. Page i can be reclaimed later when needed by a fault. In this case, θ serves as a window for lookahead, analogous to its use by WS as a window for look behind. VMIN anticipates a transition into a new phase by removing each old page from residence after its last reference prior to the transition. In contrast, WS retains each segment for as long as θ time units after the transition. VMIN and WS generate exactly the same sequence of page faults for a given reference string. Since VMIN is a lookahead algorithm, it is known that the algorithm is not practical. However, if an accurate prefetching policy is available to be able to anticipate page requirements in the time interval $(t, t + \theta)$, VMIN may be practical because, to a certain extent, the prefetching policy can distinguish which pages will not be used in the time interval.

⑥ Two–stage Selection – Another variation of page replacement policy is the two stage selection scheme. First, it select pages for discarding by any replacement algorithm and these are enqueued in the system buffer area. As a second step, some pages are selected from the buffer store for true rejection to secondary memory.

2.5 Program Restructuring

The goals of program restructuring are to improve the page level locality of reference, increase memory utilization, reduce the number of page faults, and reduce the space–time cost of executing programs. The mechanism that restructuring uses to accomplish these goals is to bring closely together in space those program parts needed closely together in time. There are two restructuring schemes according to the time rearrangement of modules: *a posteriori* and *a priori*[Hartley 88]. A*posteriori* or run–time method of restructuring performs collection, storage, and analysis of a reference string at runtime. However, the *priori* or compile–time method performs program rearranging at or before load time with information collected by the compiler from the program's source language structure.

A real program is cut up into several modules, resulting in information about the location and size of the relocatable units within the program. These programs are run on a real system to determine the reference pattern of the modules. Using a model with the original module arrangement, the reference string is run against these modules to determine the original number of page faults and resident set size. From the module information and reference pattern, a restructuring is proposed. The modules are then rearranged in the model and the reference pattern is rerun. This technique uses real programs on time sharing systems only for generation of a real reference string. By bringing together into the same group of pages those procedures and data structures that are referenced closely together in time, reordering a program's relocatable object modules can increase memory utilization, and reduce its page fault rate and space–time execution cost. Such an approach was followed by most of the work by Hatfield, Ferrari, Breecher[Hatfield 71, Ferrari 76, Breecher 89].

To improve the locality of a program, the sequences that should be taken to restructure the modules (which should be smaller than the size of a page) in a program is as follows: Firstly, the program is executed so that a reference string for modules of the program can be obtained. Secondly, using the order of reference for modules, a restructuring graph can be established. Module rearrangement attempts to make co–resident in memory those modules that most frequently reference each other. Thirdly, a static grouping scheme is applied to the graph in order to rearrange the modules according to the ordering[Breecher 89]. Thus, when module *i* makes many references to module *j*, the methodology ensures that after restructuring modules *i* and *j* will be co–resident in memory.

Ferrari categorized restructuring policies into two groups. The first methods were *nearness methods* or *strategy–independent* and did not consider the underlying memory management policy of the operating system. Instead, such methods were based on the steps described in the previous paragraph. Extensions of this technique place a module based on the reference pattern of the program during the virtual time interval $(t - T, t)$, rather than just the last reference. Because of its broader field of observation, this *extended nearness* algorithm works better than the simple nearness method[Ferrari 76]. The second method is *strategy–oriented*. Suppose two major strategies, a *priori* reference pattern of a program and a memory replacement algorithm of an operating system, are known, then there is enough information to build an effective model. The method seeks to place in memory those modules involved in forthcoming references, whilst avoiding critical references (references to different pages). For instance, the behaviour of CLRU is to place together on the same pages of memory both critically referenced modules and modules which make up the current resident set. This results in minimizing the paging rate[Ferrari 76].

The restructuring methods could get performance improvements and satisfied the necessity of an execution locating adjacently those portions of a program which are needed within a relatively short time of one another. By considering program sectors of one tenth to one third the page size, improvements in the page fault rate from 3:1 for page sizes of 512 bytes to 10:1 for page sizes of 2k bytes were obtained[Hatfield 71]. Also, the performance improvement obtainable by restructuring depends on the relative sizes of blocks and pages. In general, the smaller the blocks with respect to the pages, the better the improvement. The larger page sizes have in fact been found to increase the effectiveness of restructuring.

However, an analysis of the costs of the restructuring procedure we have described shows them to concentrate mostly in the areas of block selection, program behavior data collection, restructuring graph construction, and clustering. Gathering of referencing behavior information, a sort of preprocessing of memory tracing of execution, make conventional posterior program restructuring methods very difficult and expensive. This is because of the cost in terms of computer time which varies linearly with the number of references of the process to be examined[Ferrari 76, Beecher, Hartley 88].

Hartley investigated a *priori* restructuring scheme so that he can achieve less costly restructuring. The approach is based on an analysis of the source language structure of a program by a compiler. The code for called subprograms is duplicated and substituted in-line for the call. Ferrari also tried this static connectivity approach in performance studies involving his critical working set. In general he found that there was no page fault rate reduction in programs whose subroutines and functions were reordered according to static connectivity when compared with the program's original order. Some successful works in the *priori* structuring have been reported by Snyder[Synder 78] and

Abu–Sufah[Abu–Sufah 81]. Both of these studies achieved performance improvement by a factor of ten through rearranging subprograms and data structures, such as large arrays, rereferenced from within Fortran program loops. By reorganizing the loop structure of programs to ensure that once a page of an array is referenced as much computation as possible is done before it is replaced.

2.6 Virtual Object Memory in Smalltalk Systems

Several attempts have been made to improve virtual memory management in object oriented systems. Most of these have been done using the Smalltalk–80 virtual machine. In conventional languages, given pages in a primary memory are likely to have a significant amount of useful content in the working set. However, in a Smalltalk environment that supports dynamic or late binding, the unit of locality is a small, fine grained object in comparison to page size, and the environment is composed of a large number of these small, infrequently referenced persistent objects. This property of object oriented systems leads to degraded paged virtual memory performance. To resolve the problem Kaehler[Kaehler 83] investigated object swapping and Stamos and William reported on object grouping schemes[Stamo 84, Williams 87].

2.6.1 LOOM - Large Object Oriented Memory for Smalltalk–80 System

LOOM is a single user virtual memory system that swaps objects[Kaehler 83, Kaehler 86]. In LOOM, it is assumed that the object is the unit of locality of reference and all storage is viewed as objects that contain fields. LOOM swaps individual objects between primary and secondary memory, and it reads into main memory only those objects actually needed by the interpreter. One advantage of LOOM is that objects are assembled into groups on disk pages, so that objects which are used are brought into primary memory together. Close placement of related objects in secondary memory and cached disk pages from a pool of buffers in primary memory lower the seek rate enough so that LOOM does not need a complex mechanism for grouping.

Unlike a paging system, LOOM packs objects into main memory at maximum density because it can arrange just the right working set in main memory, and add and remove individual objects from it. However, the two different name spaces for objects in main memory and secondary memory need complicated translation code between the two representations. The object representations in both main and secondary memories are quite complex because addresses in each of primary and secondary memories have a different sized object pointer. Moreover, the translation between object representations is time consuming.

Performance evaluation of LOOM shows that object swapping is quite expensive compared to a paging system. Memory fragmentation is more common in LOOM than in a resident system, since objects not only leave holes when they die, but also when they are swapped out. Also, beyond a certain limit, adding real memory to the system will not increase its performance.

2.6.2 Static Grouping

Static grouping means any algorithm that restructures the virtual environment while the system is in a quiescent state[Stamo 84]. Object grouping is almost the same as module restructuring described in Section 2.5 which is aimed at improving spatial locality. Static grouping can be implemented by *a priori* and *a posteriori* algorithms. Rearranging related objects on the same disk page increases memory utilization and permits greater information density in primary memory. Because of the persistence of objects, it could be more efficient to use program restructuring techniques to relocate statically objects in virtual memory.

However, the differences between object grouping and the previous work on restructuring programs are caused by the characteristics of the Smalltalk system. Firstly, static grouping schemes deal with an entire programming environment composed of existing code, data, and supporting structures whereas previous restructuring algorithm manage code segments of a program. The second difference arises from Smalltalk's inability to use conventional methods for determining the important interobject references. Previous algorithms were based on the number of procedure calls, returns and nonlocal *gotos*. Such *priori* techniques were not applicable for static grouping for two reasons. First, a Smalltalk system uses small and numerous (about 17,000) objects. Treating objects as entities requires heavy computation to interpret, record, and analyze lengthy execution sequences that require substantial portions of the virtual memory system. Secondly, Smalltalk's run time binding of a procedure name (*message selector*) to its implementation (*Compiled Method*) makes control flow analysis extremely difficult at compile time[Williams 87].

The implementation of static grouping is that when a Smalltalk virtual machine loads an image it may arrange the objects in memory such that related objects are close together, thereby statically grouping objects. Depth–first, breadth–first traversals are implemented using the compiler, reference counts and dynamic statistics each to produce different initial placements. The breadth–first and depth–first algorithms view objects as nodes and pointers as directed arcs. This is quite similar to the restructuring tree described in Section 2.5. One ordering is defined by the compiler, which corresponds to an examination of an object's fields in the order assigned by the compiler when the object's type is defined. The difference between this method and control flow analysis in Section 2.3 is already explained. A second ordering is defined by static reference counts. The static reference count of an object is the total number of pointers referring to the object while the system is not in use. The third ordering was determined using dynamic statistics obtained from earlier experience with the emulator. Due to Stamo's virtual machine's inefficiency, only a limited amount of dynamic information could be obtained. So, as stated earlier, the standard techniques for nearness algorithms were not feasible for Smalltalk systems. Also, the OOZE algorithm locates all instances of a same type in one contiguous interval of virtual addresses. One odd point of Stamos's experimented result is that static reference counts and dynamic information in the three breadth–first initial placements (compiler, reference counts, dynamic statistics) and the three depth–first arrangements had little effect on either the initial placements generated by the grouping schemes or on their performance. This is believed partly to be caused by inefficiency of the virtual machine emulator that he used by the simulation.

The result of static grouping is that modifications to the basic grouping schemes shows some performance improvements. The simple and efficient grouping techniques avoided between 28 and 75 percent of the total number of page faults compare to ungrouped initial placement in small main memory sizes (less than 140K). Among the several static schemes, depth–first arrangements perform well in small main memories. However, they do not show such a good improvements for relatively large memories. Another interesting result is that grouping schemes have less of an effect on performance as the ratio of page size to object size increases. In other words, as more objects fit on each page, the detailed arrangement of information becomes less critical. One possible explanation for this decline in large memory is an imbalance in page utilization.

2.6.3 Dynamic Grouping

Object oriented programming makes use of inheritance to reuse existing code. In particular, a persistent object programming environment enables application programs to be written by inheriting from any part of a large persistent object store. In contrast, imperative languages are bound to the code that represents the program text. Also, dynamic binding makes procedure names look for their implementations at run time. These object–oriented properties lead to relatively poor paging performance on conventional virtual memory systems when compared to imperative languages and it cannot be resolved by static binding[Williams 87].

Williams[Williams 87] obtained more realistic object groupings through full memory reference tracing whereas Stamos's limited emulation used compression of the tracing. The memory reference traces include all object memory accesses, object creation and garbage collection information. It is based on dynamic information modelling in the form of LRU dynamic grouping scheme performed whilst the system is running. Grouping objects onto the page to be ejected is achieved by constructing a collection of LRU objects whose total size is less than a page. This is an implementation difference between the dynamic grouping and most of the *posterior* work which are based on program connectivity.

Dynamic grouping and some other dynamic groupings for references were simulated and Williams concluded that, for reasonable memory sizes (less than 0.5 Mbytes), dynamic grouping is always better than static grouping at reducing page faults and it changes working set more rapidly on phase transitions. However, runtime overhead is more expensive than any of the *posterior* algorithms because of the need to maintain an indirect object table, incremental copying garbage collectors, and to relocate LRU objects in a page. A total ordering of all objects in primary memory by time of last access must be maintained and it is expensive too. Also, as stated earlier, a different set of input data even for the same program can alter the trace sufficiently so that the reordered program runs worse than the original version. So it may be a waste of effort if a re-grouped file is not executed many times.

2.7 A Review of Prefetching Based System

This section reviews a number of prefetching algorithms which have been developed. These systems address similar issues to those tackled by AP. An attempt is made to compare each approach with that taken in AP.

2.7.1 One Block Lookahead (OBL) Prefetching

The alternative to grouping is prefetchings and the simplest one is OBL. As stated in Chapter one, since page $n+1$ is quite likely to be required by the executing program within a short period after the need for page n arose, an operating system fetching two consecutive pages together can reduce the number of page faults. Horspool's[Horspool 87] supports this with an example: *..if page* n *contains instructions, we might reasonably expect that control would soon flow or jump to an instruction in the next page. If page* n *contains data, it is fairly likely that the program will step through a sequential data structure (like an array) or traverse some other form of data structure that continues into the next page.*

In addition, most OBL implementations are combined with on–demand fetching scheme. This is called *on–demand OBL* and it is widely used because of its low overhead for non–random accessing of the secondary memory system. *Page clustering* and *Fill–on–Demand Klustering* in BSD UNIX are examples of on–demand OBL[Leffler 89]. Leffler adopted clustering, involves a logical page which is a multiple of the hardware page size, to reduce the cost of paging operations and their related data structures. The clustering allows fetching of as many pages as were in the cluster at a page fault. Also, Fill–on–Demand Klustering is a virtual memory operation which attempts to read any adjacent to the faulted page in the file that may also be needed. The small additional cost of prefetching pages in the cluster is compensated by the improvement in service times for future page faults.

The drawbacks of OBL and on–demand OBL, however, are that they increase the number of pages brought into the cache and they are not able to prefetch any non–OBL page on function call or long jump. OBL enables virtual memory to perform as if its page size were double since two pages are loaded for each fault. This makes memory pollution worse because there are no special algorithms to limit any unnecessary prefetching. Moreover, OBL is not an appropriate method to apply to distributed shared memory systems because such systems require a new method strictly to control misprefetching pages since the communication overhead is not negligible. Furthermore, these algorithms (OBL) are not appropriate for random access secondary memory because pages which are non–contiguous to the faulted page cannot be prefetched from the secondary store. In a random access secondary memory, the work required to fetch page i together with page $i + 1$ costs no less than the work required to fetch page i and page $i \pm n$. There are many cases caused by long jumps or function calls in a file server having a large cache memory where such fetching is required.

2.7.2 Variations of OBL

When a page has been prefetched into main memory, it is important to consider what should be done with the page if it does not get referenced within a reasonable period of time. Unless memory pollution is kept under control, it can easily happen that a prepaging policy is less efficient than its demand paging counterpart.

In Hospool's prepaging algorithm, he made use of the memory inclusion property to which OBL does not hold in order to tackle the memory pollution problem[Horspool 87]. Firstly, a variable–space prefetching policy, known as *VOBL/k*, is an approach where prefetched pages are initially given a smaller time window of *T/k*. If the prefetched page should get referenced before the time limit of *T/k* is reached, the page will be subsequently treated like a demand fetched page (and be given a new window of T time units). Thus, unreferenced prefetched pages age at *k* times the rate of demand fetched pages. So, the operating system must maintain in main memory exactly those pages whose priority values do not exceed the window. Also, the priority function determines when the imaginary references occur. Secondly, a fixed–space policy, *OBL/k*, has exactly the same form of priority function as *VOBL/k* but slightly different management for imaginary references. If a page has not previously been referenced (or prefetched), it would not be present in a memory of any size and we define its *position function* value to be infinite.

Consequently, it appears that when we suppress prepaging actions in order to avoid violations of the memory inclusion property, we are also suppressing a high proportion of memory pollution.

Like most *a priori* and *a posteriori* restructuring, Baier's[Baier 76] *Spatial Lookahead(SL)* is aimed at dynamic improvement of spatial locality. However, Baier intended to achieve predictive fetching. The notion of this method is to reduce the number of times this operation is called by transferring possibly more than one page at once, thus, the number of interruptions caused by page faulting will be diminished. Suppose a program typically executes for long periods within a locality and it generates a series of faults at intervals. Then it is possible to conjecture that recurrent patterns may exist which tend to identify the locality being accessed. For example, similar page faults could occur as the program executes in that locality. So, Baier proposed a redefinition of contiguity in a virtual address space by dynamically updating the PRED function[Baier 76] which defines logical contiguity rather than physical contiguity in an attempt to account for fragmentations.

The implementation is that the unit of virtual memory mapping ("minipages") be smaller than the unit of disk transfer, and minipages can be grouped according to dynamic reference patterns within disk pages. This affects a dynamic reorganization of minipages within disk pages. The result is effectively to prefetch related minipages on a page miss. Unfortunately, Baier's technique hardly improves performance[Williams 87] because 60% still misprefetched. Single page SL could set an improvement over LRU of about 10:1. This technique can be implemented with software and may get some reduction of page faults. However, SL still remains at the OBL level and it is inadequate for a complex prefetching system.

Since memory management on a general purpose computer is a critical operating system function, any algorithm must be inexpensive. This precludes the use of any sophisticated algorithms including many potential solutions based on mathematical programming. By this criterion, the Horspool's algorithm is impractical and only checks prefetchability for a simple lookahead block. For each page, it would need to know the times of both last real reference and its last imaginary reference for each page. This information must be kept for every page, regardless of whether or not it is resident in primary memory. Also, such an algorithm could not encompass pages other than adjacent pages. Other studies [Smith 78, Smith 87, Giraud 84, So 88, Brent 87] have addressed the problem of prefetching in cache or virtual memory systems, but with the exception of Brent's prefetching they are still concerned with OBL.

2.7.3 Cache Memory Line Prefetching

Brent[Brent 87] developed a concise program structure notation called a program skeleton that can be used for cache memory line prefetching. A source program is translated to create a machine specific cache memory prefetching control program called the *prefetch skeleton*. This is generated automatically by the compiler as it analyzes control flow and data dependency of a source program. In the sense of generating information for prefetching by compiler, Brent's work is similar to AP.

However, Brent's prefetching needs a special hardware unit runs in parallel with the CPU. This prefetching hardware unit is simulated as a simple in-cache processor. The cache machine traces CPU execution while executing the prefetch skeleton and generates cache line prefetch requests ahead of CPU demand requests. The work shows that the cache machine approach can provide some improvement both in instruction miss delays only when the specialised hardware is used with the cache.

Some problems are posed by the approach: 1) the cache machine interferes significantly with the CPU by memory contention. Total cache effect is not improved much by the contention. 2) negligible data miss reduction 3) the prefetch skeleton actually needs more memory than the actual program. These problems can be resolved by the proposed demand prefetching policy although it does not allow for parallelism or pipelining with CPU execution.

Another approach to data prefetching mechanisms was developed by Gornish[Gornish 90]. By pulling array references out from loops in Fortran programs, the data can be prefetched before control goes into the loop. Control and data dependency analyzis enables the finding of the earliest point in a program that a block of data can be prefetched. This scheme predicted 58% successful accurate data prefetching rates among candidates. A drawback is that this policy limited to only prefetching data only.

2.8 Conclusion

This chapter surveyed paging schemes, restructuring, grouping and prefetching so that we could see if any of these strategies provided facilities for forming groups of objects which can be accurately prefetched together. Most of the systems that have been developed use the *nearness* algorithm as the basic principle for restructuring program blocks. This approach, however, is not enough to support new types of secondary memory and process migration because these systems require a more accurate and randomly accessible prefetching scheme nor were they designed for object oriented systems.

The prefetching schemes discussed in the previous sections do not provide good facilities for the goal because most of them are primarily OBL based. Only Brent's and Gornish's prefetching make use of source code structure information to control the random prefetching of cache memory lines. However, the use of a special processor and its program in Brent's work place many constraints on it. Gornish's work is only concerns on data prefetching, so it is unable to prefetch program codes. If we could find a way to resolve these constraints this will be the right direction towards developing high performance virtual memory systems.

Chapter 3
AP and Object Oriented Programming Languages

The previous chapter has discussed the back ground to virtual memory systems. It also surveyed some related work with prefetching. In this chapter, an intra–object and inter–object relationship is built by making use of the properties of object oriented systems. To support the prefetching of object pages, each object requires the addition of links to related objects that are individually addressed.

There are two aspects to providing prefetchability, the first is the construction of a relationship between objects using properties of object–oriented languages. The second is the building of another relationship between function calls by conventional control flow analysis. The purpose of this chapter is to describe the former approach of building an intra and inter object relationship for prefetching pages for the objects. Also, how control flows among objects in the same hierarchy is described in this chapter. The advantage of object page prefetching is that it can be used for general purpose prefetching of objects or pages in a variety of memory hierarchies.

The chapter begins by describing a formula for accurate prefetching. The following sections discuss how intra and inter object relationship can be made using encapsulation, inheritance and other object–oriented properties. The section after discusses the requirement for object data prefetching in a UNIX environment. The final sections deal with the problems of establishing the relationship at compile time and run time.

3.1 Formulation of AP

An accurate demand prefetching uses two memory fetching policies at the same time to fetch the pages of a process when a page fault occurs: prefetching for necessary pages in the near future and on–demand fetching for a faulted page. In prefetching, more than one pages are fetched by anticipation of the process's future page requirements whereas in demand fetching only the page referenced is fetched on a miss. So, the behavior of the *i*th process r*i*(*k*) is the number of the page containing the virtual address references of the process *Pi* at time *k*, where $k = 1, 2, \ldots$ T measures the execution time or virtual time. The set of pages that P*i* has in main memory just before the *k*th reference is denoted by a resident set {z*i*(*k*–1)}, and its size (in pages) by z*i*(*k*–1). A page fault occurs at virtual time *k* if r*i*(*k*) is not in the resident set Z*i*(*k*–1) [Hwang 84].

So, under the assumption of on–demand fetching for a faulted page and prefetching for the object pages associated with the faulted page, Z*i*(*k*) is as following;

$$Zi(k) = \{zi(k{-}1)\} + \{ri(k)\} + \{ri(k+n)\} - Q(k)$$

In fixed space replacement systems, the first term on the right hand side Z*i*(*k*–1) is the set of pages that *Pi* has in main memory just before the *k*th reference. The second term is a demand fetched page for which a fault occurs at virtual time *k* and which will be accessed after time *t*. The third term is a set of prefetched pages {r*i*(*k*+*n*)} where *n* is a number of prefetched pages at time *k*, $n = 1, 2, \ldots$. The final term Q(*k*) is a set of replacement pages chosen by a replacement policy (see Section 2.4.1) when a set of pages {r*i*(*k* + *n*)} are prefetched. Notice that among the replacement algorithms, MIN or VMIN can be realized practically for Q(k) in the formula. Since AP anticipates necessary pages accurately, it may also distinguish pages which will not be referenced in the near future.

3.2 The Influence of the Properties of OO Languages on AP

The properties which should be considered in building a relationship for prefetching are data abstraction, encapsulation, inheritance, dynamic binding, construction and destruction of objects. This section describes these properties in detail from a view point of prefetching and the following sections discuss how the properties can be used for AP.

3.2.1 Data Abstraction and Encapsulation

An object is a basic element which has an internal state and operations. To construct objects, object oriented languages provide data abstraction and encapsulation. Data abstraction supports the construction of objects considered abstractly by both the implementor and user of a class in terms of their behaviour rather than their state. An object is an abstraction that has data and functions, with the functions being defined by a set of operations that are available on the object. The operations and the internal state of the object are defined by a class declaration, so that objects are instances of a class, and all instances of the same class share identical functions. So, data abstraction allows the separation of the abstract behavior of a class from its implementation details.

A set of functions provided by a class provides the only means by which instances of the class may be manipulated. When an object is used the user does not care about the internal structure of the object because it is not necessary to know how the functions are implemented or how the class is structured. For instance, an object that represents the int_Stack in Figure 3.1 (the pointer to integer bottom, top and current) could have operations which enable the stack to be pushed and popped by maintaining the pointers. How the actual stack is maintained as the internal state of the object changing need not concern the user of object, who only needs be concerned with the behaviour, defined by the two operations provided by the class. This is the power of data abstraction, since the implementation of the abstraction is divorced from the behaviour that the abstraction provides.

Encapsulation is a basic feature of object oriented languages because of its facility for data hiding and protection. Also, it is an important factor for demand prefetching of an object's data because each object class specifies and tightens an abstract data type.

A class that represents an integer stack using an array is illustrated in Figure 3.1. In the class int_Stack, the internal status of the stack is implemented as three pointers to an integer array: bottom, top and current. The int_Stack is represented by the **class int_Stack** which provides two operations to push and pop an integer to and from the array. An alternative implementation of the class using an array is to use linked lists. Thus, although the internal data structure of the stack is changed, the interface of the operation push and pop would not need to be changed. This is an example showing that data abstraction and encapsulation provides a consistent interface without affecting users of the class in spite of changes to the internals of a class.

```
  class int_Stack{
     int *bottom;
     int  *top;
     int  *current;
  public:
     int_Stack(int size){
        bottom = new int[size];
        top = bottom + size;
        current = top;}
     ~int_Stack(){
        delete bottom;}
     void push(int Int){
        *(current++) = Int;}
     int pop(){
        return (current>bottom) ? *--current : 0;}
     void exception(){
        //exception handling routine }
  }
main()
{
  int_Stack a(100);
  a.push(5);
  b = a.pop();
  cout << b;
}
```

Fig.3.1 The class int_Stack

Encapsulation provides a good property for constructions intra object relationships in the AP system. Because the property enables us to tie the data structure and operations together so that once a part of the object is referenced the potential that other parts may be referenced is high. The simplest application of the intra object relationship is object data prefetching. It happens when a member function is invoked but its data object has not resident in memory, then the data should be prefetched. For example, it occurs in **a.push()** in the Figure 3.1. When the member function **push()** is fetched the encapsulated data object **a** can be prefetched at the same time. This is based on the data dependency graph which is explained in Section 3.3.

3.2.2 Inheritance

Inheritance is another property which enables the features of an existing class (base part) to be re–used by a newly declared class (derived part). This property is assumed to be provided by object–oriented languages. It is possible to eliminate the reimplementation of shared code in a class hierarchy if inheritance is used when designing a new class. It enables new classes to be derived from existing classes, with the new class inheriting the data and member functions of the existing class. So, inheritance provides programmers extensibility and code sharing. There are two ways to refine the existing class: by adding extra functionality or by providing a restricted interface to the inherited data structures. In a number of systems this takes the form of a class hierarchy in which common functionality is shared between classes that belong to the hierarchy [Dixon 88].

When a new class is derived from a base class it can inherit the attributes of the base class. The base class may also be termed the super–class of the derived class, and the derived class sub–class of the base class. A class hierarchy is pictured in Figure 3.2 as a model of the object layout of a base class and a derived class. The class Employer is derived from its base class Person. The object Fred consists of base object attributes and

```
class Person {
private: char *name;
        int id.;
        int birthdate;
public:  ...
};
class Employer : public Person
{
private:  short position;
          int salary;
 public:  ...
};

Person Fred;
Employer John;
```

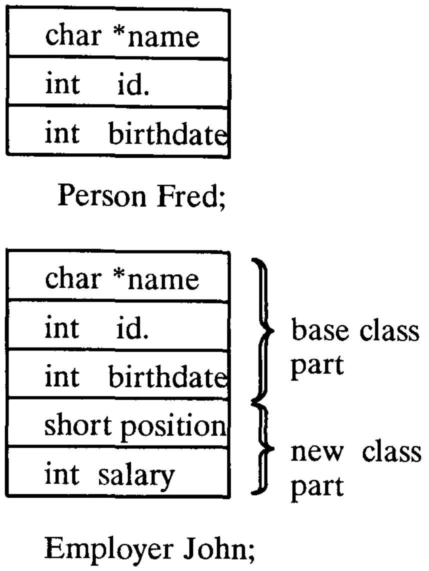


Fig.3.2 The *employer* class hierarchy

the attributes of the derived object itself. The inherited attributes for Person are left unchanged in the derived class and the new class provides additional attributes to the new class so that they are more applicable to the derived object.

There are some constraints to refine the existing class in some object oriented languages. In Smalltalk-80[Goldberg 83] and Objective-C[Cox 86], they should inherit all data and operations from all of their ancestors or nothing because they do not allow partial inheritance of operations. This mechanism complicates object interfaces as the hierarchy becomes deeper and forces some redundant operations to be included in a derived class[Parrington 88].

If a derived class is only allowed a single base class then the language used to declared the derived class supports sub-typing inheritance e.g. Smalltalk-80, Figure 3.3(a) shows this sub-typing, or single inheritance: class C directly inherits only from class B. When a class is allowed more than one immediate base class then the language supports multiple inheritance e.g. C+ +: in Figure 3.3(b) class C directly inherits from both class A and class B. More complex arrangements are possible. For example, Figure

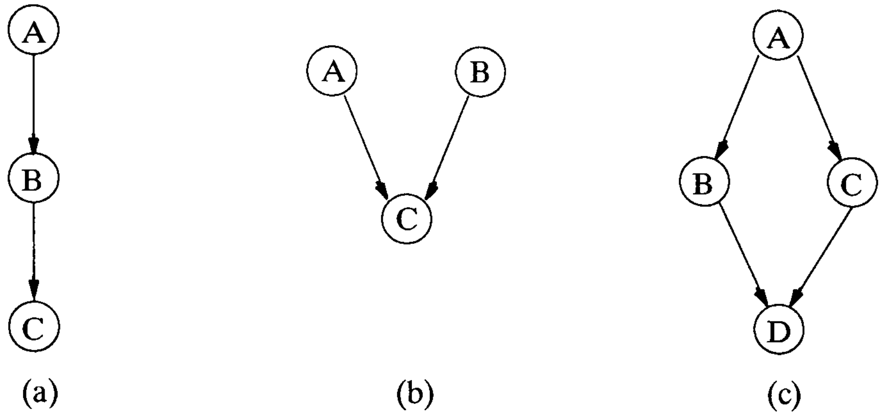


Fig. 3.3 Single, multiple and repeated inheritance

3.3(c) demonstrates repeated inheritance where the multiple inheritance paths from the derived class D lead to a common shared ancestor A [Meyer 88]. If a new type can have more than one parent type then it can inherit the operations and instance variables from each of them. However, repeated inheritance introduces a semantic ambiguity: in the example, should an instance of class D have one set of instance variables for its ancestor instance of classs A, or two sets ? C+ + provides the keyword *virtual* to specify that only one instance of the repeated base class is to be inherited.

AP can makes use of single inheritance as well as multiple inheritance for the sake of object prefetching by building inter object relationships for the objects in a hierarchy. Multiple inheritance is more useful for AP because it ties more objects together thus increasing the logical locality of reference. The details of using inheritance for AP is described in Section 3.3.

3.2.3 Dynamic Binding or Virtual Functions

When a function which is mentioned in a derived class is called to perform an operation on an object the definition actually used is determined at execution time according to the class of the object. Thus, if an instance of the base type is expected as a parameter to some operation such as a pointer to a member function, then an instance of the derived type can be supplied instead. This implies that the object cannot simply be treated as being of the base type, rather a lookup must be performed at run time to determine the actual type of object supplied so that the correct version of the subtype is actually called[Parrington 88]. In most cases the compiler can detect the type of the object and ensure that the correct version of subtype is invoked when required but dynamic binding objects are exceptions. This is called dynamic binding and it enhances flexibility through runtime binding of operations to objects. Dynamic binding encourages placing the code that deals with a particular class of object in the implementation of the object's class rather than in the client program thereby making the client program more general.

For example, if we need a type that contains a list of arbitrary type of objects, this generic list can be built easily if it is designed so that types which are inserted into the list are declared to be derived types of some base type and thus can be inserted into a list with ease. If a main program trys to print out descriptions of all objects in the list, the program simply selects each entry in the list and invokes the printing member function of that entry. Since the compiler cannot detect what type of object will be on the list, the determination of which particular implementation of the subtype member function to call has to be made at run time.

C++ is a strongly typed language with early-binding (at compile time) of operation names to the code that implements them. However, as noted above, there are some occasions where dynamic binding must be used, otherwise objects could not be treated as instances of their parent type and passed to operations that expected them to behave as instances of their parent type [Parrington 88]. In C++ this is known as the virtual function mechanism.

The implementation of dynamic binding is based on the procedure call mechanism and the mechanism is described in C++ manual as follows: "The interpretation of the call of a virtual function depends on the type of the object for which it is called, whereas the interpretation of a call of a non–virtual member function depends only on the type of the pointer denoting that object." Usually, this is implemented using pointer to member function. Thus, if the type of the operand is X, the type of the result is "pointer to X". Since any type can be substituted for the X the operator & in C+ + is polymorphic.

Most late binding object oriented programming languages such as Smalltalk–80, Guide[Balter 89] and Objective–C support dynamic binding and, in particular, the binding mechanism is associated with message passing in Smalltalk–80. Since a message is sent to an object to perform an operation in the object, the message contains an object's name only. The object which receives the message selects an appropriate operation for the request at runtime. In Smalltalk terms, this run time binding is carried out between an object name (procedure name or *message selector*) and its implementation (*Compiled Method*).

From the prefetching point of view, dynamic binding makes it difficult to build the relationship between operations (member functions) and instance variables (object data) at compile time. For example, func_A() is defined as a virtual function in class **Base** and its derived class **Derived** as well. A pointer *p* to the base object can point either a base object or a derived object. Then the real object which pointer *p* points to in the virtual member function called p– > funcA() cannot able to be ascertained at compile time. Thus, a call to func_A() must determine at run–time which particular implementation to invoke based upon the type of the object currently under consideration. So, building the relationship for prefetching is resolved by run–time AP. This point will be described in detail in the following subsection.

3.2.4 Construction and Destruction

The mechanism which provides memory space construction of an object which consists of some data structures and operator can be categorized into two groups. These two memory allocation strategies for execution of a program are static and dynamic allocation. Allocation that occurs when a program is compiled is static memory allocation. Otherwise, if a reserved free space for a program execution is allocated for objects at run–time it is dynamic memory allocation. An object's *extent* is defined as an object's lifetime, that is a period of time which storage is bound to the object while a program is executing[Lippman 89]. Whenever a new object is created, a constructor of that class is called. For C++, there are three distinct object creation methods: automatic, static and free store.

- A static object is created when the program starts and it will be destroyed at the termination of the program. Variables defined at file scope are said to have static extent. Storage is allocated before program start up and remains bound to the variable throughout program execution. For the initialization of these static objects, Stroustroup says "No initializer can be specified for a static member, and it cannot be of a class with a constructor"[Stroustroup 86]. This could mean two things: Firstly, if a class has a constructor, that class may not have static members. Secondly, you cannot have a static data member which needs a constructor. Thus, a static data or a static constant data member is allowed, as long as the member does not require a constructor or the member is private. Similarly, a destructor does not have any control over what will happen to the memory occupied by the object it is destroying after the destructor is finished. A relationship for prefetching of static object can be established but it can not contain constructors or private data.

- An automatic object is created by the constructor of the class each time its declaration is encountered in the execution of the program. The life of an automatic object is similar to that of local variables. Variables defined at local scope are spoken of as having local extent. Storage is allocated at each entry into the local scope: on exit, the storage is freed. Automatic objects do not retain their values from one call to the other because the objects are newly created whenever the function is called. As far as prefetching is concerned, this kind of object cannot be prepared for prefetching at compile time and even then they are not worth prefetching because local variables on the stack are unlikely to make frequent page or object faults.

- Objects allocated on the free store are spoken of as having dynamic extent. Storage allocated through the use of the operator *new* remains bound to an object until explicitly deallocated by the programmer. Explicit deallocation is achieved by applying the operator *delete* to a pointer addressing the dynamic object. The *new* operator handles dynamic memory allocation from a free unallocated memory space given to a program that it may utilize during execution. This kind of object can not be managed properly in AP because its address is not known at compile time. However, if the relationship is allowed to be built at runtime and if the constructor and destructor are extended to fills in the addresses of objects into the prefetch table whenever they are created and deleted, these objects in the heap can be prefetched.

The philosophical basis of the *new* mechanism in C++ is that allocation and deallocation are completely separate from construction and destruction. Construction and destruction are handled by constructors and destructors. Allocation and deallocation are handled by operator *new* and operator *delete*. At the time a constructor is entered, memory has already been allocated in which the constructor will do its work. Here is a simple case:

```
void f(){
       T x;
}
```

Executing f causes the following to happen:

```
Allocate enough memory to hold a T;
Construct the T in that memory;
Destroy the T;
Deallocate the memory.
```

Similarly, `T* tp = new T;`
does the following:

```
Allocate enough memory to hold a T;
If allocation was successful,
    construct a T in that memory;
Store the address of the memory in tp
```

and `delete tp;`
means:

```
If tp is not null,
    destroy the T in the memory addressed by tp;
    free the memory addressed by tp.
```

3.3 *Yoyo* in Objective-C and Smalltalk-80

High inter-object control flow in the object hierarchy in Objective-C and Smalltalk-80 is obvious. This point is well described in Taenzer's paper[Taenzer 89]. In a late binding object oriented language, a complex problem behavior is implemented by methods in a class. Objective-C and Smalltalk methods sends *self* and *super* messages in their object hierarchy to implement required behavior. These messages cause frequent control flow within a class hierarchy. In Smalltalk, for instance, a *new* message is sent to the metaclass (class object) which returns a new instance of the class. Then, an *initialise* message is sent to its new instance. A set of classes in the hierarchy define only the initializing method and inherit the *new* method from the class. This makes it hard to understand when this initialize method will be used. In this case, you must find the *new* method in the superclass (or its super-superclass or its super-super-superclass, etc.) and discover that it sends itself the initialize message. Furthermore, in writing an initialize method, you have to remember how to send the initialize message to super objects[Goldberg 83].

The control flow of messages on these methods in the same hierarchy is described as a *yoyo* problem by Taenzer. Because in Objective-C and Smalltalk the object *self* remains the same during the execution of a message. "Everytime a method sends itself a message, the interpretation of that message is evaluated from the standpoint of the original class (the class of the object). This is like a yoyo going down to the bottom of its string. If the original class does not implement a method for the message, the hierarchy is searched (going up the superclass chain) looking for a class which does implement the message. This is like the yoyo going back up. Super messages also cause evaluation to go up the class hierarchy." Thus, the yoyo is a problem caused by software reuse because when writing a new classes most of its methods are derived from its base class. However, as far as prefetching is concerned, the yoyo can be used for object prefetching as a means of providing a strong relationship between the inter-class hierarchy. Note that Smalltalk-80 treats everything uniformly as objects, including fundamental data types and blocks of code. There is no separation between code and data objects. So, object data prefetching which is describing in the following section is not necessary in Smalltalk-80.

The *yoyo* can be graphed so that nodes represent straight-line code fragments and data objects and arcs represent procedure calls, returns, (conditional)branches, and data accesses. A standard technique for determining the frequently used arcs is to interpret the execution of the program and maintain a traversal count for each arc. As explained earlier, such control flow analysis techniques are not feasible for a system containing a large number of small code and data objects. This was explained in detail in Section 2.6.2 static grouping.

3.4 Intra and Inter Object Relationships for AP

The properties of object oriented languages described in the previous sections give many influences to control flow in the languages. These can be described in terms of control and data dependency inside an object and among objects in the same hierarchy or in different object hierarchies.

The modularity in most object oriented languages is designed to support function sharing and structured programming. Since the methodology of object oriented programming is to divide a system into a set of objects, which closely match the concepts of the real world problem, providing a way of managing the complexity of the programming task. This kind of data abstraction and information hiding enables modular design in programming[Parrington 88]. This modularity influences high locality of reference by control flows between objects.

Memory reference patterns for object oriented programs are more localized than for similar programs using traditional models. In the case of the Amber[Chase 90] object/thread model, the body of an object operation can reference only the thread stack and the contents of the object itself, so an executing operation is likely to make a sequence of memory references local to the current object[Chase 89]. Locality in a data abstraction programming language like CLU[Liskov 86] has similar characteristics.

However, the problem of a persistent object oriented programming environment, in particular in Smalltalk-80 or Guide[Balter 90], is that locality of reference is neither bounded by contiguous segments of code nor operating on some data because most of application to reuse any portion of a large persistent object[Williams 87]. Persistent objects have a lifetime which is independent of that of a program or the process that uses them. The system provides a permanent address space for these objects. This can be viewed as a substitute for a traditional file system. So, the address space containing these persistent objects tend to be large and it imposes a requirement on the supporting system for efficient object management. In particular, if a system supports the *persistent programming*[Balter 90] approach, giving users a unifying view of the system, the system should support long-term storage units. Therefore, a system supporting persistent programming may have many garbage objects in memory and these may decrease the locality of reference of the program. Persistent object oriented virtual machines have strong small spacial locality of reference within an object but global locality of reference is rather low because a number of these small spacial localities are dispersed over the whole object range.

This is slightly better in an early binding language like C++ because process execution is bound in just one executable file. Locality of reference for objects in the same branch of a class hierarchy is quite high compared to that for objects in different branches. But still the locality in C++ is not enough for a high performance virtual memory system because it supports dynamic binding. Thus, it can be said that object oriented programming languages have two characteristics with regard to locality of reference. Encapsulation and inheritance enhances locality of references but dynamic binding lessens it.

Several cases can be found in C++ showing high locality of reference. Firstly, a derived class is inherited from its base class, so the encapsulation is still preserved between two objects. Thus, a member of a derived class has no special permission to access private members of its base class. When a member function of a derived class needs data access to its base class this should be done by a function call with scope. Secondly, any particular instance of dynamic binding is always restricted to a particular inheritance hierarchy. Thirdly, an initialization process for derived objects depends on its base class if there is some subtyping. For instance, let us look at class declarations like this:

```
class Color { /* stuff */}
class Primary_Color : public Color {/* more stuff */}
class Blue : public Primary Color {/*still more stuff */}
```

Here, when we call "new Blue" this causes calls on the constructors for Color, Primary Color and Blue executed in that order. When we call "delete b" it invokes the destructors of Blue, Primary_Color and Color in that order. Once control goes into a boundary of the class hierarchy to create an object at a leaf of the hierarchy the control would go up to the base class to create and initialize the object and down one hierarchy to the other until it reaches the leaf object itself. Once an object is created then object data access will be followed by the member functions in the objects because the data object is only able to be modified by the member functions. After manipulating the data, when control leaves the member function some of the object will be cleaned up by calling destructors in the hierarchy in the same sequence as for construction.

These series of invocations are made whenever an object is created, used and destroyed. It means that a series of function invocations form a spacial locality, so, it can be said that there is a relatively high tendency to locality of reference in a class hierarchy by calling functions within it. Although these objects have high locality of reference, they may possibly spread over many different pages.

To enhance logical locality of reference for these dispersed objects, the objects can be logically grouped using encapsulation and inheritance, the logical group prefetched when part of it is invoked. The operations and data structures encapsulated by a class can be retrieved from a source program at compile time, and kept in a relationship table. An object name is mapped into a class name and this can be recorded in the table. The inheritance tree can normally be deduced by the compiler. Thus, invocations which may result in yoyo can be identified, and the inheritance tree can be recorded in the table. If this is performed during the parsing phase, all the relationships between objects in a program can be identified by the compiler. This is similar to the attempts at *a priori* program restructuring which were based on static program connectivity.

Where encapsulation ensures locality of reference is high within the object an attempt can be made to prefetch all member functions and data at one go. Some exceptions from accurate prefetching may arise. For instance, if we consider the exception handling routine in the class **int_Stack** in Figure 3.1 five member functions will be prefetched together whenever any one of the member functions is referenced. However, although the potential of referencing all four member functions is very high, the exception handling function will not be invoked unless an exception occurs.

Inheritance shows a high locality of reference between objects in the same class hierarchy. However, as far as accurate prefetching is concerned, there may be some member functions which are not involved in the yoyo phenomenon e.g. another exception handler in a base class. Therefore, not all member functions in base objects need be prefetched since it is unlikely that these functions will be referenced in the near future.

In conclusion, intra and inter object relationships can provide high locality of reference, and they can be used for object grouping. However, when all member functions and object data related to these intra and inter relationship are prefetched, some prefetched functions may be not referenced and they may pollute the memory. Imprudent prefetching of the logical group can cause memory pollution, thus, prefetching ought to be combined with control flow analysis of the logical group.

3.5 Object Data Prefetching at Compile Time

Object data which is encapsulated by member functions could be prefetched in prospect of a text page's future requirements. This is significant in an operating system where object code and data reside in different pages because fetches for these pages are incurred independently and these causes more faults. There are two possible way to suppress the faults by building the relationship between object data and code: using the object–oriented properties with either data or control dependency analysis. Object data prefetching by data dependency analysis can be built either at compile time or at runtime. Building the relationship at compile time is not as simple as at runtime and as is shown below. This section describes how the object code and data may be separated in UNIX and the following section discusses the methods for prefetching object data.

3.5.1 Separation of Object Data from Code in UNIX

Most of UNIX systems support several executable file formats. In executable type 407 files, instructions and data are intermixed but a 410 file is pure executable and a 413 file which is pure demand–paged executable, instructions are separate from data. Process text and data images in 410 and 413 occupy separate sections of memory in a certain executable file formats which are shown in Figure 3.4. There are several advantages of keeping text and data separate: protection, sharing and the fact that the data segment may grow during program execution.

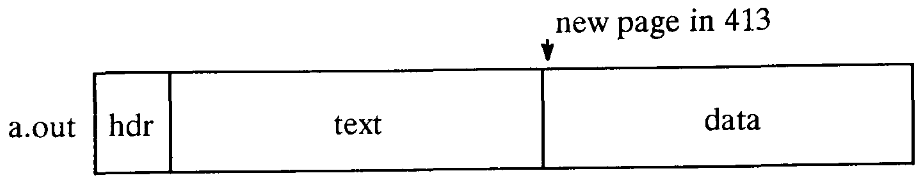


Fig.3.4 Executable file types for 410 and 413

However, because of this separation of text and data in executable files other problems are generated. For example, text and data fetchings are performed independently. When a process invokes a non–resident member function which suppose to have an object data it causes a text page fault and the process will stop execution until the page will be read into main memory. When the page is fetched the process will restart but it will face another page fault because the object data which the function required is not read in yet. So, the data page loading is done in the same way as the text page. In this situation, an object data page related to the object member function can be stucked together and then they can be fetched simultaneously.

Figure 3.5 illustrates the model of object storage in the UNIX operating system. A virtual address space is separated into a manageable size page, for fetching and purging. These pages are again grouped into regions: code, data(static data and heap area) and stack. Most of the statically and dynamically created objects are stored in the data or stack reagions. So, the object data in Page $M+1$ is not able to coexist in the same page as its member function which is stored in page M. In this environment, the method of stucking the member function and object data together and prefetching them at the same time are described in the following section.

3.5.2 Object Data Prefetching with Data Dependency Analysis

To suppress page faults, control and data connectivity can be used to relate pages and to move them in and out of main memory together. Among the properties of object–oriented languages, encapsulation has the dominant effect on the data prefetching policy. Object data prefetching is closely associated with data dependency analysis too. Since data–flow analysis is affected by the control constructs in a program, this property is used to build the relationship between them. In fact, the data analysis can be done in the level of statements and this is explained in detail in the following sections.

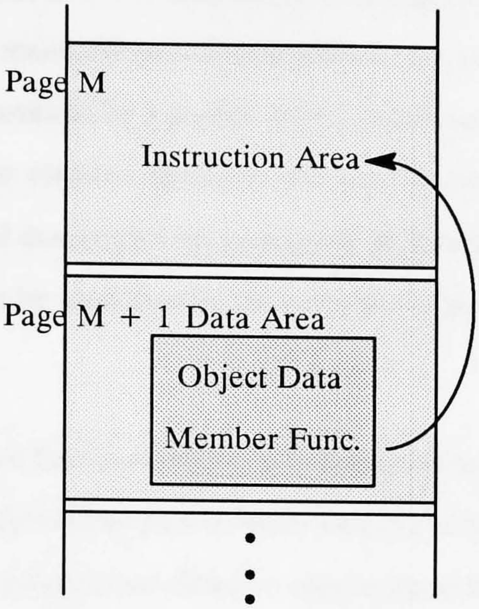


Fig. 3.5 Object data and codes are stored in different pages

3.5.2.1 Establishing a Relationship between a Member Function and its Object Data at Compile-time

In a demand paged virtual memory, when a non-resident address is referenced the address is used by the fault handler to read the faulted page into main memory. On the other hand, to run a prefetching system where some fetches of pages into the main memory before any reference to those pages had occurred, some *priori* reference information for imaginary faults should be stored in a table which provides references so that the pages are prefetched correctly. This can be provided by establishing an intra object relationship table which can be built if we make use of encapsulation properties described in the previous sections.

Member function and its object data can be retrieved from a source program at compile time and be kept in the relationship table. Object data can be linked to associated member functions quite simply if we make use of the object invocation syntax.

Member variables of a class in C+ + are referenced using the **.** or the **- >** operator in the same way that structure members are referenced in C. For instance, pushing an integer onto a stack object **a** is invoked by **a.push()**. In this statement, the relationship between object variables and their member functions can be defined directly at compile time. If this operation is proceed during parsing a program all the names of member functions and their object data can be identified by the compiler so as to record them under their own class names.

However, there are four constraints on building the relationship at compile time. The first is dynamically created objects as described in the previous section. In C+ +, there are three possible storage allocations for objects according to the way in which they are created. Firstly, consider names with global scope, when machine code is generated by the compiler, the position of each objects data relative to a fixed origin such as the beginning of an activation record must be ascertained. Information about the storage locations that will be bound to the names at run time is kept in the symbol table (name and address or offset). This information in the symbol table can be used to index the object data. Secondly, local names whose storage are allocated on a stack are supported by the runtime system[Aho 86]. The positions of automatic and free store objects are known dynamically at run time except for those of static objects described in Section 3.2.4. So, only statically declared objects can be prefetched if the relationship table is built at compile time. However, the relationship table can be implemented for free store allocated objects if object addresses in the relational table are filled in by an extended *new* operator whenever it creates new objects at run time.

The second obstacle to implementing object data page prefetching at compile time is dynamic binding. Conventional programs have statically bound procedure calls, whereas bindings in some object oriented languages are often performed at run time. This means that it is not feasible to determine the precise body of code that may be used by an application[Goldberg 83, Stamo 84, Williams 87, Taenzer 89]. Therefore, the properties of late or dynamic binding and dynamic object creation are severe constraints on building the relationship at compile time.

The third constraint is the many–to–one relationships between the member functions of a class and instances of that class (i.e. objects). In particular, the relationship between polymorphic functions and object data is many to many at compile time. If the relationship is built at compile time looking up a many–to–one or many–to–many table for each object is expensive and it is unlikely to be practical. One potential solution to this problem is to group these polymorphic functions in a page or adjacent pages statically to increase spacial locality of reference.

The fourth obstacle is caused by the use of pointers and aliases. A call of a procedure with *x* as a parameter (other than as a value parameter) or a procedure that can access *x* because *x* is in the scope of the procedure. We also have to consider the possibility of "aliasing," where *x* is not in the scope of the procedure, but *x* has been identified with another variable that is passed as a parameter or is in the scope. Another case is that an assignment through a pointer that could refer to *x*. For example, $*q = y$ is the assignment of *x* if it is possible that *q* points to *x*.

3.5.2.2 Prefetching Object Data at Runtime

Constructing the relationship table at compile time to prefetch dynamically created and bound object pages is not so simple. The relationship is a static description of objects and does not include objects specified during execution. Thus, the serious weakness of the relationship with respect to such data is the inability to prefetch dynamically bound functions and data whose addresses are not known at compile time. There is, however, an alternative which could be adopted quite simply at runtime. Thus simple method adds little overhead to the execution time of a process but is quite limited a scheme because it makes use of the functions in a symbolic debugger.

When a page fault happens on a function call, the arguments to the function (including the object pointers) are already on the stack. Even when a virtual function which is multiply redefined in derived classes is called to operate on an object, the selection actually used is determined at execution time based on the class of the object. The arguments to these dynamically bound functions are already specified and the invocation of a member function is now the same as a procedure call. The first argument is a pointer to the object mapped to **this** in the member function. When a fault handler tries to read the member function which caused the invalid address into memory the related object data pages which the member function will need in the near future can be prefetched using the first argument. Suppose the arguments are referenced by pointers, the object data may reside on the heap and the pointers to the object data are on the stack. So, the object data can be prefetched indirectly using the pointers and the rest are the same as the non-pointer method.

The implementation details of this are described in Section 5.6.4. So, as described in the previous section, object data prefetching for dynamically bound functions can be resolved. In a conventional virtual memory system, the object's data might have been read into memory during execution of the called member function by another page fault for the object. Thus, the potential page faults can be suppressed by prefetching the object's data.

Current implementations of object data prefetching can resolve most faults caused by static objects even they are referenced by pointers or aliased. Although prefetching object data on the stack is not required, if object data are referenced by the pointers on the stack, they should be prefetched properly. Run-time creation of objects by the *new* operator and variables which addresses are computed and specified at runtime can be prefetched by this method. One reason to be able to achieve the data prefetching is that a symbolic debugger provides most of addresses required and resolves aliases and pointers.

There are some constraints remaining on object data prefetching at runtime. As Brent states, all variables cannot be prefetched either by the use of a prefetch table or by this runtime object data scheme. This is because there are register variables and local variables which are defined within activation records in a stack. Stack variables do not have fixed addresses that are known when the program is compiled. However, the stack is not a large component of the miss ratio since it is accessed often with good locality and the referenced parts of the stack do not tend to get replaced. Therefore, stack variables do not need to be prefetched often except during phase transition.

3.6 Conclusion

This chapter described how AP makes use of source program structure information, in particular, the properties of object-oriented languages. Data abstraction and encapsulation enable us to build relationships between operations and their data. Inheritance provides another means of tying together objects in the same hierarchy because control flow within the hierarchy is very common. This is shown because object oriented programmers take the approach of reusing software by inheritance and this causes a vertical control flow. On the other hand, dynamic binding makes it difficult to build the relationship for prefetching at compile time.

A practical method of achieving the accurate object data prefetching was proposed by using encapsulation. From the foregoing we can see the possibility of using the properties of object oriented programming languages for prefetching object data. However, the constraints and exceptions which we mentioned do not allow us to make a general accurate prefetching scheme, because all encapsulated member functions and data in an object and all constructors and destructor functions in the same hierarchy are not guaranteed to be referenced unless there is direct control flow between them. Therefore, the next chapter introduces control flow analysis to reinforce the relationship.

Chapter 4
Prefetching Blocks

The previous chapter considered the effects of the properties of object–oriented languages on accurate prefetching. It has been assumed that encapsulation and inheritance allow the creation of inter and intra objects relationships. This chapter introduces an approach that helps to achieve the goal by making use of conventional control flow analysis for C++. The relationship considered in chapter three to prefetch objects can be reinforced using control flow analysis because the object oriented properties are expressed explicitly in a program.

Control and data dependency analysis are major areas of interest in compiler study in respect of optimum code generation and parallel processing. The flow graph of a program is used as a vehicle to collect information about the intermediate program for common subexpression elimination, dead–code elimination and renaming temporary variables and interchange of statements. Data dependency graphs represent the data dependency structure of a program. The data dependency structure influences the cost of partitioning the program for parallel execution. In addition, control and data flow analysis are able to be used for object prefetching.

The use of the *basic block*, which is used for code optimization in general compiler theory, is the first step in building a program control graph for prefetching. Then the basic blocks are combined to form a prefetching block to represent single–entry single–exit block. Some unnecessary inner branches and inner loops in basic blocks are eliminated in a prefetching block to simplify and optimize the prefetching table. Also, data objects associated with a prefetch block are tied to the block.

The first section shows how it is possible to decompose a program into basic blocks for building the basic units of a control flow graph. It then considers some of the primitive graph symbols used to represent a program compactly. The section after discusses how to make a prefetch block using basic blocks and the primitive graph symbols. To decompose high level language statements into their low level structures before being graphed the following two sections tackle the branch and loop statements of C++. The following sections then discusses adding object data to the prefetch block. Also, some characteristics of a prefetch block in terms of page faults are addressed in this section. The final section assesses the technique developed in this chapter for constructing the prefetching block by comparision with similar work.

4.1 Program Decomposition

To establish a prefetching block, a program should be decomposed into basic units of computation and flow control. The prefetching block contains some objects which are possibly dispersed in several pages. A decompose–and–merge algorithm is used to build the prefetching block. So, the fundamental units of computation and flow control are discussed in this section.

A program consists of one or more statements that are branches, assignments, copies and function calls. Simple assignment and copy statements are mainly computational details. As far as control flow is concerned, the statements can be classified into two groups: branch statements or computational statements. To show the control flow of the program, a complicated program consisting of branch and computation statements can be simplified if all redundant computational details are removed. The technique of using a graph based on *basic blocks* can be found for prefetching in cache memory[Brent 87] or other applications such as parallel processing[Baxter 89, McCreary 89, Montenyohl 88]. Structural graphs of programs concisely characterize the execution paths of the program without the confusion of extraneous information.

4.1.1 Basic Block

To analyze control and data dependency of a program in compiler theory, the program should be decomposed into a set of basic units. A *basic block* is defined as the unit which is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without a halt or the possibility of branching except at the end[Aho 86]. If there are branches or other exceptions these become leaders of new independent basic blocks.

Basic blocks are often used for a graph representation of three–address statements. Also, if a set of basic blocks are linked by flow–of–controls, this directed graph is called a flow graph. This flow graph which is generated from a basic block is used for improving and optimizing code generation. Various code optimizers, for example loop optimization and dead code elimination, try to use such transformations to rearrange the computations in a program in an effort to reduce the overall running time or space requirement of the final target program.

4.1.2 Three Address Statements

Three address code is a sequence of low level statements including some statements for flow of control. They are quite fundamental statements to form basic blocks. Aho[Aho 86] summarized the types of common three–address statements as follows: (For a complete description of the three address statements, see[Aho 86].)

- Assignment statements with a unary, binary arithmetic or logical operation.
- Copy statements
- Unconditional jumps
- Conditional jumps

- Function calls
- Indexed assignments
- Address and pointer assignments

4.1.3 Building a Basic Block

A basic block is an elementary node of a prefetching block which contains location information with regard to object variables, sequential arithmetic or logic expressions and statements including function calls. As stated above, these objects can be located in anywhere in the address space. The basic block is a unit to tie them together in a sequential block for prefetching.

To form a basic block, three address statements for unconditional, conditional jumps or branches are not included in a basic block. Instead, these *lead* a new basic block. There is a general method to build the basic block. We first determine the set of leaders, the first statements of basic blocks. And then for each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program. The rules to determine a leader are as following.

- The first statement is a leader.
- Any statement that is the target of a conditional or unconditional goto is a leader.
- Any statement that immediately follows a goto or conditional goto statement is a leader.

As stated earlier, the nodes of the flow graph are the basic blocks. One node is distinguished as initial: it is the block whose leader is the first statement. The rest of the flow graph is linked from the initial node with directed arcs. The general rule for establishing a whole flow graph is that there is a directed edge from block *B1* to block *B2* if *B2* can immediately follow *B1* in the following execution sequence:

- if there is a conditional or unconditional jump from the last statement of *B1* to the first statement of *B2*, or
- *B2* immediately follows *B1* in the order of the program, and *B1* does not end in an unconditional jump. Here, *B1* is a predecessor of *B2* and *B2* is a successor of *B1*.

These general rules for building a control flow graph based on basic blocks are adopted for making a prefetching block and eventually to create a prefetch table. However, the basic block involved in this section is not very useful without modification because the three–address statements which are elements of the basic block are too low level and have fine granularity. So, elimination is performed to make a practical size of prefetching block in the following section. Before we discuss the prefetching block, five basic symbols are defined so as to be able to build a control flow graph in high level language statements.

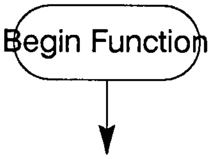
4.2 Control Flow Graph of a Basic Function

To provide a prefetch table to the AP system, a concise description of the structure of a program and its prefetchable unit, called a prefetch block (PB), is developed in this section. The description does not have any semantics of a language but concisely shows the syntactic control flow graph of a program. Conventional graph techniques are used to analyze programs. However, a method which is more simple, easily generated at compile time and easily used by AP must be developed. So, the flow graph is generated automatically from the source program at compile time. A C+ + compiler is extended to generate the flow graph and prefetch tree during the code generation phase.

Program structure can be graphed using only the following set of basic symbols: *begin function*, *return*, *sequential block, call and object data* and *branch*. However, to represent the prefetchability of a block or function, a symbol for *prefetch block* appears in the program graph. These graph symbols are extended from Brent's models[Brent 87] to fit for object oriented system and they are defined as follows:

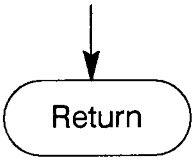
4.2.1 Begin Function

The entry point of the function. There can only be one begin function symbol in a function. This unique start point of a function can be decomposed into more than one three address statements because it includes formal argument passings. However, a function calling three address statement except argument passings is matched to this begin function. The object data are not shown here with the function start symbol because a member function have more than one objects to manipulate. So, it is graphed with the function call symbol. The symbol for *begin* function is an ellipse shape.



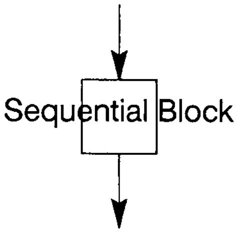
4.2.2 Return Function

A function can have one or more exit points where control leaves the function and object manipulation is finished. Each exit point of a function is shown as a return. The symbol for return is same as for start, an ellipse. From a prefetching point of view, a return address, the location to which the called routine must transfer after it is finished, is important information. The return address is usually the location of the instruction that follows the call in the calling procedure. In a small memory space, the page having a function return point can be purged out if the nested function is very deep.



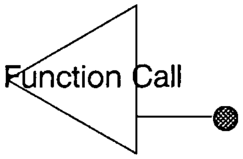
4.2.3 Sequential Block

This is a sequential execution path of code with a single-entry and single-exit point. A block can be classified in two ways. The first is the sequential block which has a single entry single exit point. The second is the nonsequential block which contains some internal branches. For our purposes a block can contain more than one statement and even branches, as long as the single–entry single–exit rule is maintained. However, function calls are not allowed to appear within sequential blocks because a function call statement is an independent graph for further processing. One difference between a sequential block and a basic block is that a basic block consists of three–address statements but a sequential block is based on high level statements rather than three address spaces. The sequential block symbol is a rectangle.



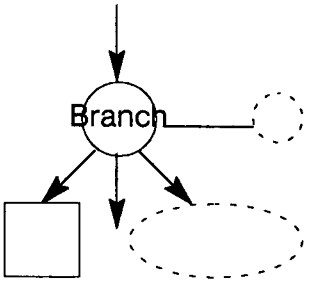
4.2.4 Function Call and Object Data

The invoked function name will be shown inside a triangle of which one corner is open. The open side is linked to the caller and the other side to the called function. The called procedure can be substituted in place of the call. Object data which are encapsulated in a class with the member functions are graphed together with the function call so that it can be prefetched. Also, object data which are associated with the function are shown together, because, whenever an object member function is called the member function has encapsulated object data. So, this associated object member function and its object data can be graphed together. We implicitly assume that the objects are passed to the member function by value or by pointers, so the object data graph can be set up at the member function level.



4.2.5 Branch

The conditional branch symbol identifies a point where two or more possible paths through the code can be taken. The branch symbol is a circle with several arcs as outputs. Each output is a links to the beginning of another sequential block in the graph, which is called the branch target. Since the graphs represent only the structure of control flow, the conditional test that is implied by the branch need not be shown. Also, the branch point is the beginning point of a prefetch block. So, the branch point contains prefetching information such as the addresses of objects in the prefetch block.



The reason why conditional branches are so vital in control flow analysis both in basic block and a prefetching scheme is that they break spatial locality of references. For example, if two function calls are being associated with each block in a conditional branch and the two functions are physically located sufficiently apart from each other to break the spatial locality of reference, the branch test at the branch point decides which function to invoke following the branch. The unconditional branch had an important meaning in building basic blocks using three address statements but it is not very important when building high level control flows for prefetching. Since most unconditional branches can be predicted their control flow, even if they can break spatial locality of references, can be managed by prefetching.

4.3 Prefetch Block

The basic block which computes a set of expressions can be represented as various types of three–address statements. Three–address statements can be combined into larger units in three ways: sequencing, conditionals and loops. Sequencing is achieved simply by writing non–jump statements one after another. The non–jump statements include various types of assignment and copy statements. The function call statement is a sort of non–sequencing control flow since the control transfers to and from the callee function naturally. However, it still obeys the single–entry–single–exit rule if the function does not have nested function calls associated with a conditional branch. Although it breaks spatial locality, this can be managed by prefetching some related pages. A program can, therefore, be seen as a set of basic blocks and branches which link the blocks in a sense of control flow. A*prefetch block* consists of a set of instructions between an entry and an exit point. For the purpose of building a prefetching block, a prefetch block may contain more than one statement and even a nested loop or internal branches as long as the single–entry–single–exit rule is maintained.

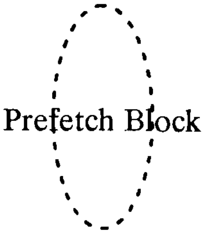
One difference between a basic block and a prefetch block is in function call statements and inner branches. The definition of a basic block does not include function call statements in its sequence of consecutive statements. However, the prefetch block does includes function calls. One difference between a prefetch block and Brent's[Brent 87] "Execution block" is that the latter does not allow function calls to appear in execution blocks.

A block which has neither conditional branches nor function calls can be defined as a *primary prefetch block,* otherwise, it is defined as a *non–sequential block.* A primary prefetch block consists of a set of instructions between a branch instruction and the next following branch instruction. As far as prefetching is concerned, a primary prefetch block does not have significant meaning because it does not includes any locality breaking object codes but it still has some prefetchable objects on the heap or stack. Compared to a primary prefetch block, a *prefetch block* can contain a variety of prefetchable objects such as long jumps, member functions and object data, with the exception of branch points. Non–sequential block does not exist practically because it is decomposed into primary prefetch blocks and prefetch blocks.

If we take a look at the C+ + grammar in Figure 4.1, a primary prefetch block consists of only a sequence of statements like the production from (3) to (6) in which flow of control enters at the beginning and leaves at the end without halt or the possibility of conditional branching except at the end. A prefetch block can have nested sequential blocks. A prefetch block does not have conditional branch statements but has function calls, i.e., does not have productions from (7) to (14). A function consists of data declaration parts and a set of blocks. Thus, a function can be classified in the same way as a block. A function which has only prefetchable blocks is a prefetchable function.

The reason why functions and blocks must be classified into subblocks is that it is an important criterion for judging the prefetchability of a function or a prefetch block. The control flow of a program which consists of simple sequential functions and sequential blocks can be anticipated straight forwardly. Thus, if a program does not have any conditional branch, its control flow would correspond to a depth first traversal of the activation tree that starts at the root, visits a node before its children, and visits children at each node from left to right order[Aho 86, Horwitz 88]. However, a big constraint on code page prefetching is conditional branches in a program (productions (7) to (14) in Fig4.1). For instance, a nonsequential block (a prefetch block) with no function calls but inner branches must be prefetched by a system which reads a whole function code into memory before execution but a nonsequential block which has function calls and branches could not be prefetched even in that case.

The graph of a prefetch block is similar to that of a sequential block. A prefetch block can include a sequential blocks in the control flow graph because the sequential blocks obeys the single–entry–single–exit rule. This prefetch block is symbolized as a dotted ellipse.



4.4 Control Flow for Conditional Branches and Loops in C+ +

The C+ + grammar for a function, block, conditional and unconditional branch is shown in Figure 4.1. These high level statements can be converted to three address statements which are similar to assembly code[Aho 86]. For example, the "for" and "while" loops are high level statements for easy programming and they are not well matched with the symbols in the previous section. Since these high level statements in a source program can expand to more than one branch in assembly languages. The following is a list of C+ + statements that directly affect the execution paths.

These high level language statements do not correspond to the symbols in the previous section. So, these statements must be decomposed into their inner structure using the symbols. They are as follows:

1 *Call* statements, *proc–name(arguments):* This directly corresponds to the symbols of *function call* and *object data.* Object data is a part of the argument passed to the function but it is an entity to be prefetched.

```
function-dcl    : decl;                                          (1)
att-function-def :type   decl    arg-dcl-list    base-init  block
function-def    |decl  arg-dcl-list    base-init   block ;
stmt-list       :stmt-list   statement
                |statement ;
block           :{ stmt-list } ;
simple          :e
                |BREAK | CONTINUE | RETURN e | GOTO Id           (2)
                ;
statement       :expr   ;                                        (3)
                |simple SM                                       (4)
                |att-fct-def                                     (5)
                |block                                           (6)
                |IF condition statement                          (7)
                |IF condition statement ELSE statement           (8)
                |WHILE condition     statement                   (9)
                |FOR  (   statement   e   ;   e  )  statement    (10)
                |SWITCH   condition   statement                  (11)
                |ID  COLON  statement                            (12)
                |CASE   e    COLON   statement                   (13)
                 DO   statement   WHILE   condition              (14)
```

Fig. 4.1 C+ + grammar for function, block, conditional and unconditional branch

$\boxed{2}$ *Break* is a sort of unconditional branch out of the current switch block or a loop. So, this can be graphed with a directed arc. *Continue* is similar to Break, i.e. this is a branch to the beginning of the current loop and can be graphed as an arc to the start of the loop. *Goto Id* directly corresponds to an arc to the label.

$\boxed{3}$ *Return* statement corresponds to the symbol in 4.2.2. It can have a return value or not but the return value is of no interest from the prefetching point of view, because mostly it is stored on the top of stack which is very likely to be located in main memory.

$\boxed{4}$ *If* statements (statements (7) and (8)) are straightforward to graph. They have a conditional branch symbol followed by an arc directed to a subgraph. The branch test statement is not limited to simple statements but it can contains calls or sub branch statements.

$\boxed{5}$ *While* condition test expressions have a loop test and a branch test as well. There are two arcs: one to a loop body subblock followed by a loop back and the other is its exit.

$\boxed{6}$ *For* statement

```
for(expr1; expr2; expr3)
    statement

is equivalent to
expr1;
while (expr2) {
    statement
    expr3;
}

nb. if statement contains a break, expr 3 is never executed.
```

Most commonly, expr1 and expr3 are assignment expressions or function calls. Expr1 is the initialization code that is performed before the loop. Expr3 is the loop index increment statement that is executed at last of every loop. Expr2 is a relational expression[Kernighan 78]. The statement is the body of the while loop. So, *for* statements have four subgraphs which can contain any statement or symbol.

$\boxed{7}$ *Switch* statements (statements (11), (12) and (13)) : The switch statement is a special multi way decision maker. The conditional expression matches one of a number of constant values and takes a branch to the one that matches. It can be decomposed into if statements and graphed accordingly.

$\boxed{8}$ *Do* – While statements : The loop body is executed first and followed by a conditional branch test. The body subgraph and conditional statement subgraph are serialized in control flow and followed by a branch symbol.

For example, Figure 4.2 shows a simplified set of class hierarchy definitions and a set of prefetch blocks. The derived class **Blue** declaration for the base class **Color** is illustrated. The main() function has an object creation and an if–then–else statement with some member function calls in the block. The main program can be graphed using

```
class Color { ... };
class Primary_Color : public Color { ...};
class Secondary_Color : public Primary_Color{...};

class Blue   : public Primary Color{ ... };
class Purple : public Secondary_Color{ ... };

main(){
      blu = new Blue;
      ppl = new Purple;
      if (clr != 0) {
            .....
            blu.paint();
      }
      else{
      ...
      ppl.erase();
      }
}

Blue::paint()
{ Primary_Color::draw(); }

Puple::erase()
{  Secondary_Color:: clear(); }

Primary_Color::draw()
{     ...
      while(c=next())
            c->draw();
}

 Secondary_Color::clear()
 {
   ...
  }
```

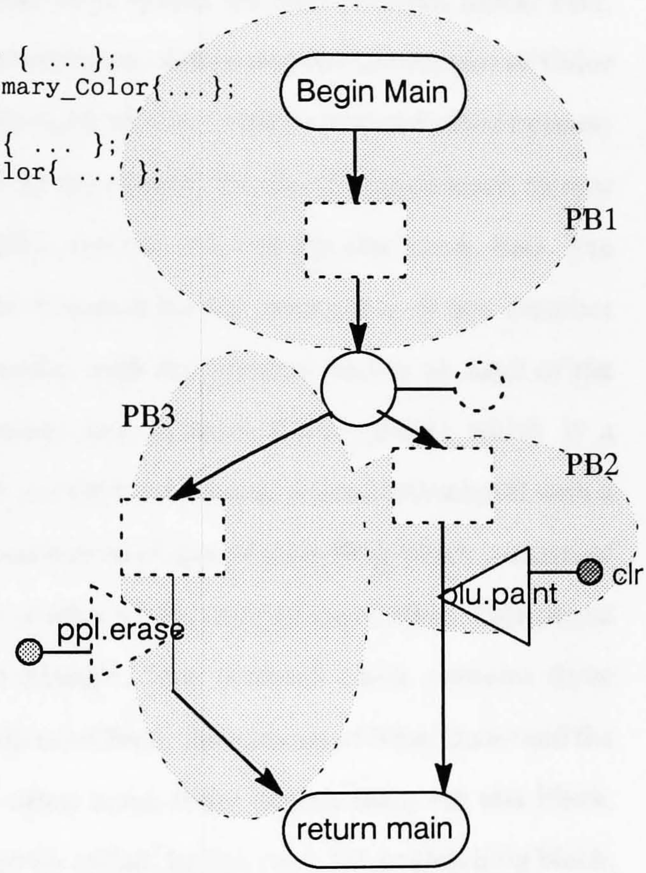


Fig. 4.2. Program frame and prefetch blocks

the symbols defined above in terms of its control flows and object data dependency. The prefetch blocks starts with the Begin_Main symbol followed by a subgraph concerning two object creations. When the main function is called, the first prefetch block, PB1, contains the address of builtin_new, the constructor of **Blue** and the constructor of **Color** because it is a base class of **Blue**. The prefetch block is not able to stretch further because the **if** branch is followed by the creation of the object. So, the **if** branch leads to new prefetch blocks: one in the **then** block, PB2, and the other in the **else** block, PB3. The head of PB2 has some sequential blocks followed by the invocation of two member functions. These are graphed as two triangles with the function names on each of the triangle. **blu.paint()** has a nested function call Primary_Color::draw() which is a nonsequential function because it has an another conditional branch associated with a nested function. Also, the member function has an object of class **Blue** which consists of one base class, **Color**, plus the members unique to the derived class, **Blue**. This object graph, a shaded circle, is linked to the triangle. This prefetch block contains three important prefetching data items: the address of Blue, the address of Blue::paint and the address of Primary_Color::draw. On the other hand, if the branch takes the **else** block, Ppl.erase(), Primary_Color::clear(), would be called. In this case, the prefetching block, PB3, contains the address of an object Purple which contains all the base and derived parts of the data, namely, the address of function erase() and function clear().

4.5 Adding Data to the Prefetch Block

In earlier chapters, we suggested that building programs using the object-oriented method was efficient, because it concentrates on modeling entities from the real world as related logical objects. This method leads to the interesting notion that individual objects should be responsible for fetching their encapsulated parts, particularly when the objects are persistent. Thus, encapsulation provides a means to prefetch object data while the member function is read into main memory. As

Brent[Brent 87] mentioned, data misses are so significant in virtual memory paging or cache line prefetching system because these are typically the major component of faults, so, a prefetching scheme that uses a control flow analysis notion is not complete unless data is prefetched along with instructions.

As stated earlier, an object is an instance of a class and consists of some data and a set of member functions that determine the external behavior of the object. The class of an object specifies what operations may be applied to the object data, because the member functions provided by an object have access to the instance variables and can modify the data. Therefore, the relationship between object data and member functions can be built so as to prefetch one of them when either of them is referenced. Thus, when object X's data is fetched from a disk to some arbitrary location in main memory, related member functions of the object can be moved at the same time and vice versa. The object may also have more data, referenced indirectly. Pages for indirect code and data cannot be prefetched when the object is first accessed. The reason for this is discussed in detail in the following section.

The prefetch block provides an accurate representation of the flow of control of a program and contains some information on object references. For example, it can contain addresses of instance variables, member functions and long branch points. However, not all data references can be contained in the prefetch table because AP is not aware of the existence of pointers and references. For example, in indexed assignments of the form $x = y[i]$, x is set to the value in the location i memory units beyond location y, so it is hardly reasonable to expect to have precise addresses of objects referenced by indices or pointers at compile time.

Object data prefetching which is described in Chapter 3 can be adapted to the control graph and it is shown in Figure 4.2.. There were two approaches suggested for acheving this goal: compile time analysis and run time analysis. However, if object data prefetching is intend to be combined with the prefetching tree, it should be done by the compile time strategy and the run time scheme used additionally for objects which cannot be prefetched any other way.

4.6 Prefetch Block and Paging

The prefetch block is now a complete description of the program structure and its data accesses. It identifies the branching structure, procedure calls, and some of the references to data in a program. Many possible uses can be found for this prefetch block. Analysis of the program structure is facilitated by using the control flow analysis since only significant structural information is retained in its concise format. Object code and data prefetching can now be performed using the prefetch block since many of the necessary data references are specified.

In order to create as big blocks as possible, the prospective sequential blocks or functions can be merged into a prefetch block. A prefetch block was defined as one or more sequential blocks between a conditional branch and the following conditional branch in the control flow. The key features of the prefetching scheme using prefetch blocks presented here are as follows:

- Control flow cannot be predict at a conditional branch, so it should rely on demand prefetching at that point.

- All objects' data associated with member functions in a prefetch block are prefetchable if they are static at compile time. Otherwise, they can be prefetched at run time by the method described in the Section 3.5.2.2.

- A set of adjacent primary prefetch blocks or prefetch blocks in a control flow can be merged to form a bigger prefetch block. Low–level assembly language branches which build primary blocks are not significant for most memory prefetching or program analysis. One of the benefits of this approach is that it includes the significant information, while unnecessary information is excluded.

- No page faults would happen in a prefetch block in a pure prefetching method, however, they are likely to occur in a demand prefetching.
- A minimum, viable prefetching block is a prefetch block which has statements dispersed over at least two pages.
- A prefetch block in the shrinking phase should be built independently because it is independent to that in the growing phase. However, the distance between these two can be a good parameter for the selection of a replacement page.

To resolve the naming problem for prefetching blocks which occur at every page fault, a possible solution with hardware is that a register can be allocated to contain the name of the current executing prefetching block by tracing CPU execution so that it can provide the block name to the fault handler. This could be a fast but expensive approach because of the hardware support required. Another approach taken in this thesis is software based using the functionality in a symbolic debugger. The naming is resolved in the simulation by looking up the prefetch table to find a prefetch block which is associated with a page fault. The lookup of a faulted function can be performed efficiently by hashing the function names and then finding the prefetch block in the function that uses the faulted address. By this method, naming problems can be resolved because a faulted address is uniquely mapped to a prefetch block. The details of implementation are described in chapter 5.4.

Consider again the control flow graph shown in Figure 4.2 which has already been described in Section 4.4. In Figure 4.3, suppose the graphed code located at page N and some library codes invoked in the prefetch blocks such as malloc() are dispersed in pages $M+1$ and graphic control routines which are invoked by draw() are stored in M. The

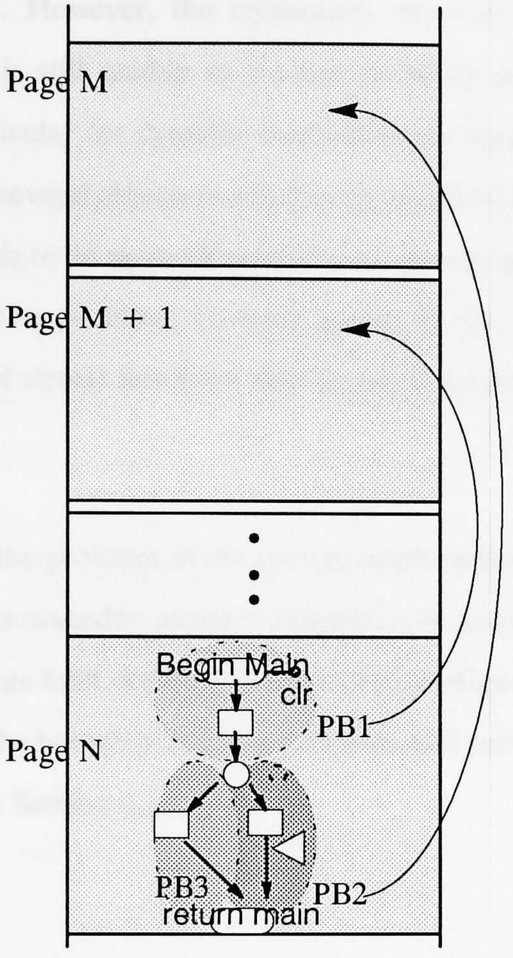


Fig.4.3 The dispersion of object codes

branch points, if it is a leading point of a prefetch block, contain all necessary addresses which would be essential information for prefetching the pages. After checking the branch condition, one of the prefetch blocks which may be in page M or in M+1 would be selected and prefetched. At the moment, all loops in the control flow graph are omitted for simplicity but some optimization will be applied to the looping later on.

Most page faults caused by function calls in nonlookahead virtual memory systems can be suppressed by prefetching a prefetch block. In OBL, some proportion of useless prefetches could occur but these kinds of misprefetching will be reduced and page faults which are caused by long jumps could be suppressed by AP. It also suppresses much memory pollution. However, the constraints described in Section 3.5.2.1 are still effective and AP is still unable to manage properly faults caused by the language properties. In particular for dynamic binding(virtual function), in the case of taking a uniform action for several objects in a hierarchy by calling the virtual functions defined in each class, it is likely to be more efficient if some grouping policy is introduced for these virtual functions. The problem, however, posed by this method is that a page which contains a group of virtual functions may be not referenced, and then, it contaminates main memory.

Moreover, the problem of the current implementation is that AP is only able to suppress page faults caused by member function calls and their objects. But natural page faults, such as a page fault occuring between two contiguous pages cannot be managed properly because the branch prediction problems still remained. This point is described in greater detail in Section 6.3.

4.7 Comparision with Similar Work

It is worth comparing AP with other object migration schemes although AP has different basis to them since it has been developed for efficient virtual memory and process migration based on paging scheme in the current implementation. Some object migration schemes, for instance SOS, have a uniform migration mechanism for local and remote accesses. Their first goal was to implement a one-level storage, transparently integrating the so called "vertical migration" (to and from disk) with "horizontal migration" (between memory contexts). Actually, vertical importation from storage into a context is identical to horizontal importation, therefore, this comparision emphasizes on how AP can be applied to distributed object oriented systems.

4.8.1 SOS

SOS extended the object concept to distributed or "fragmented" objects which shows a single object externally by providing local fragments or *proxies* which are distributed internally[Shapiro 89]. A fragmented object is a group of *Elementary Objects* which is dispersed but it is represented by local fragments. A fragment can add new fragments to the group and group membership is preserved across migration. In SOS, an object is mapped into a context which is an address space. It may contain any number of objects which have their own unique identifier. To allow object migration, SOS provides two different identities: an address and a location–independent reference (containing an object identifier). An address is not meaningful outside of its instantiation context. It needs to be explicitly translated into a reference in order for it to be embedded in a message. This permits pointers automatic conversions between references and addresses.

The object migration mechanism in SOS is quite simple. When an object is to be migrated from source to destination context, it moves prerequisite, object data, object code and then reinitializes it in its new site. Code and prerequisite are migrated recursively if not already present in the new site. Because SOS migrates the prerequisites on demand, the migration time is significant. It therefore provides static groupings where a group is created when a proxy provider migrates a proxy to another context. The details of static grouping are not known but SOS does not have a facility for accurately prefetching objects.

4.8.2 Emerald

Emerald[Jul 90] is an object–oriented language for distributed programming, featuring fine–grained mobility. Mobility in the Emerald system differs from existing process migration schemes in two important respects. The first is that it is object based so the unit of mobility can be much smaller than in paging based process migration systems.

Second is that there is language support for mobility such as *join*, *leave* and *attach*. The compiler transforms the user-defined object representation in order to facilitate migration: its first few bytes are a standard descriptor and all fields of a similar type are grouped together. Conceptually all objects live in a single, network-wide address space. An object reference is global, but a local reference is optimized into a pointer.

Another important point when moving objects containing references is deciding how much to move. Suppose an object is a part of a graph of references – one could move a single object, several levels of objects, or the entire graph. The programmer can specify movability explicitly and group related objects together. This is the difference between AP and Emerald, thus, AP groups objects to be migrated by compiler transparently to the user.

4.8.3 Guide

Guide[Balter 90] is an object-oriented distributed operating system which provides persistent objects and concurrent computation through threads. To support persistent objects, the system supports a permanent repository for objects, as a substitute for a traditional file system. Guide objects imply two meanings: one is storage units and the other is instances of a class. All guide objects are persistent and they exist as long as they are referred to by at least one other persistent object. These objects have system wide location independent unique identifiers. The execution abstraction is called a job which may be viewed as a multiprocessor virtual machine consisting of distributed concurrent activities operating on objects.

Since objects are stored in multiple sites and location transparent secondary storage, they are loaded on demand into a virtual memory for execution by an *object fault*. A basic object invocation is similar to procedure call but it is called an *object invocation*. An invocation is specified by a reference to an invoked object, the name of a method and the parameters of the invocation. When an object is not found in main memory where the execution takes place, on demand fetching for the object takes place. Object management like this can be a good application area for AP because small objects which are stored in different disk blocks can be managed more efficiently together rather than individually. Moreover, if AP is adapted to the Guide system, the naming issue for every persistent but location independent object can be resolved by using the object index table which contain names and addresses of objects in the system.

As far as object migration is concerned, Guide has a different concept from other systems. Whenever a new object is created, the object image is replicated to every node in the distributed system. In the case of a diskless workstation, the image is moved to the machine at boot time. It, therefore, does not require object member function migration but only object data. However, if we assume that the image is not moved to the diskless workstation by brute force, AP can prefetch some related member functions according to the object id and function name.

4.8.4 Comparision with AP and Discussion

As we can see in the above, AP has good functionality for managing object migration in such systems as SOS, Emerald, Guide and others. In these systems, object page fetching and migration based on accurate prefetch in AP is more efficient than individual on demand object migration. A significant advantage of the AP approach is in performance. As stated in Chapter 2, the pure object-oriented approach for memory management such as in SOS and Emerald is expensive. Each *elementary object* in SOS has a size of 50 to 100 bytes and up. All migratory objects have system descriptors which cause considerable overhead to the system managing them. These overheads for managing individual objects could be reduced if the systems adopted AP for their object storage management, in particular, fetching a group of objects at the same time for virtual memory management and object migration.

For instance, the migration algorithms of SOS and Emerald use simple grouping schemes. When an SOS object is to be migrated to a new context, all system descriptors for the object's data are copied and, in particular, the object code (operations) is imported recursively. Thus, SOS transfers all code objects by brute force or on demand fetching (always using deep copy[Sollins 79] and therefore losing shared behavior) and has some static groupings for special occasions such as moving the name server when booting. Emerald also provides facilities for forming groups of objects which will move together as if they are linked each other. In comparision to AP, SOS and Emerald remain conventional migration strategies because they do not provide any user transparent accurate prefetching for groups of objects.

Another notable point is that AP provides object data prefetching by intercepting the arguments passed to a member function. This technique depends on the argument passing schemes chosen, such as pass–by–value or pass–by–reference. In particular, argument prefetching in distributed systems is quite important because otherwise serious performance problems could arise. In a distributed object–oriented system such as Emerald, Clouds or Guide, the desire to treat local and remote operations identically leads to the use of the same semantics. On a remote invocation, access to an argument by the remote operation is likely to cause an additional remote invocation (call back[Jul 89]). The references must be resolvable on all nodes with uniform semantics – the local–address / global–reference distinction can exist but it should be hidden from programs by providing a single, network–wide address space, and compiler support for trapping remote access[Chase 1989]. For this reason, systems such as Argus have required that arguments to remote calls be passed by value, not by object–reference. Similarly, RPC systems require call–by–value since addresses are context dependent and have no meaning in the remote environment. In any of these cases of argument passing schemes in local or remote calls, object data management through proper prefetching arguments could help to reduce the object data fetching time.

4.9 Conclusion

We have seen that the use of control flow analysis to reinforce the relationship between objects is feasible because some inter–object and intra–object relationships described in the previous chapter are expressed explicitly in object invocations. The prefetch table describes source code structure, in particular, it concerns itself with function calls and static object data in a concise structural representation.

The prefetch table consists of several prefetch blocks which are separated by major branch points to represent an accurate description of the objects of a program. Consequently, the prefetch block enables us to group objects and function calls in the block together and prefetch them at the same time. We discussed how this methodology can be useful for object migration in object–oriented distributed computing systems.

Chapter 5
Implementation of AP

The previous chapters described how objects' member functions can be collected together and prefetched at the same time so as to reduce the number of page faults and increase global paging system performance. This capability is provided by establishing a prefetch table of an object's function calls based on control flow analysis and data referencing patterns. This chapter discusses in detail one way in which the accurate prefetching mechanism can be implemented.

The approach taken in this implementation falls into three major parts. The first is the construction of a virtual memory system simulator which is a simplified model functioning for the AP system. The second part is establishing the prefetch tree. This is generated by an extended C+ + compiler. The prefetch tree generation implementation is based on the concepts described in the previous chapter. The final part is the building of an interface which enables AP to be run using the prefetch table on a simulator. Rather than providing an elaborate model or an actual implementation on a real machine, AP was simulated to provide a prototype for testing its feasibility.

This chapter begins by describing the accurate prefetching virtual memory system model. The section after describes how each module in the model is simulated and can be used during the operation of the system. In particular, this section shows the details of how the prefetching manager and the page fault manager operate for accurate prefetching.

The following sections describe the extended compiler in more detail. In particular, these sections describe how the notions for prefetch tree generation described in the previous chapters are implemented and consider the steps that are required to complete the prefetch table.

We then discuss naming issues for prefetch blocks when looking for a relevant prefetch block with a faulted address. The section after describes in detail AP's running operation which consists of three phases. The final section summarize and discusses the technique developed in this chapter for implementing accurate prefetching.

5.1 The AP System Model

The basic architecture of AP is shown in Figure 5.1. The approach taken in this simulation makes the whole virtual memory operation clear and simple enough to show how the accurate prefetching of pages works. The layered structure on the bottom of the figure shows that the whole simulated AP system works on top of UNIX and is considered as an application program from the UNIX point of view. The AP virtual memory simulator consists of two major parts: the prefetching virtual memory simulator and a prefetching table generating compiler. To simulate the address references of a process, the virtual memory simulator consists of five modules: a code executor, main memory (s_main) and secondary memory (s_secondary), prefetch queue manager and page fault manager.

First, the virtual memory simulator provides an instruction executor to the simulator to trace the control flow of the program and evaluate all addresses involved. Main memory and secondary memory are simulated to provide an address space for processes and these are accessed by the executor. Main memory specifies a virtual address space for use by a process as primary memory. The secondary memory also specifies an address space so as to contain a whole process image. Both of the memories are assumed to allow random access to any page. This is because AP is not limited to just disk based secondary memory systems but a local use or a remote RAM disk or remote file server as well. The key module is the prefetch manager which handles accurate

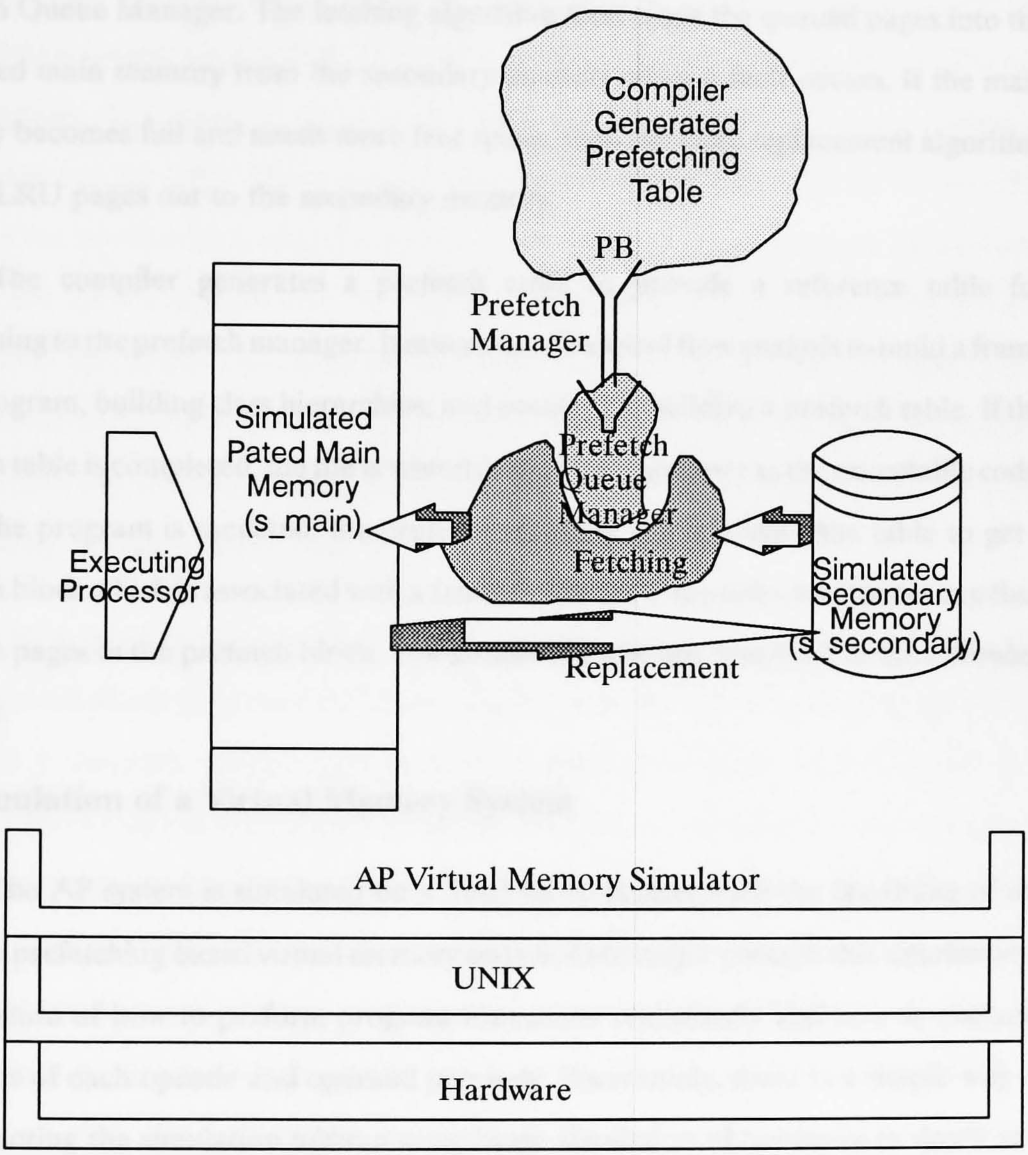


Fig.5.1. The structure of AP system

prefetching by searching a prefetch table and obtaining related pages in a prefetch block at the same time at a page fault. The prefetch manager gets its prefetch information from a prefetch table generated by the compiler and these are stored in a prefetch queue by the Prefetch Queue Manager. The fetching algorithm then reads the queued pages into the simulated main memory from the secondary memory when a fault occurs. If the main memory becomes full and needs more free space, then the page replacement algorithm purges LRU pages out to the secondary memory.

The compiler generates a prefetch table to provide a reference table for prefetching to the prefetch manager. It makes use of control flow analysis to build a frame for a program, building class hierarchies, and eventually, building a prefetch table. If the prefetch table is completed, the file is stored in the same directory as the executable code. When the program is executed, the prefetch manager will reference the table to get a prefetch block which is associated with a faulted address. The prefetch manager can then prefetch pages in the prefetch block. The details of these are described in the following sections.

5.2 Simulation of a Virtual Memory System

The AP system is simulated on a Sun3/60 to demonstrate the feasibility of the accurate prefetching based virtual memory system. One major point in this simulation is the question of how to perform program executions realistically and how to evaluate addresses of each opcode and operand precisely. Fortunately, there is a simple way of implementing the simulation without complicate simulation of hardware in detail in a virtual memory system. Instead of all details of computational, control signals and data transfers on buses in a computer system, we are just interested in program control flow and address references. This simple but efficient simulator can provide a main parameter - fault rate - to measure the performance of virtual memory system. The functionality of each manager and object prefetching scenario are as follows.

5.2.1 Executor

The executor simulates the CPU's memory accesses and it has two major functionalities. The first is that it traces the control flow of the CPU in a virtual address space when it executes a program. The tracing is exactly the same as that of the CPU of a real machine. The machine code in the real main memory provides an execution environment to the CPU. The approach taken in this simulation is that whilst the CPU executes the machine code step by step and computes instructions as usual, the executor follows the CPU's control flow by tracing the program counter. To control CPU execution of the program in step mode, i.e. instruction by instruction, the executor causes the processor to stop or to continue its execution by inserting a break point between every instruction in real memory. In fact, the breakpoints are related to the ptrace() system call provided by the operating system.

The second function is that the addresses referenced by each instruction are evaluated and checked by the executor to see if they are in a given address range. While the CPU is stopped, the executor performs address evaluation of the opcode and operands for the current instruction. Whenever the code executor executes an instruction the address evaluator fetch the addresses of the opcode and operands of the instruction in order to determine if the reference was a valid or invalid memory access. The evaluation of the accessed opcode address is relatively simple compared to that of the operand because the opcode is matched with the contents of the program counter of the CPU. On the other hand, operand address evaluation is not so easy. Operands have basic addressing modes: direct, deferred, indexed, register and combinations of these modes. MC68020 microprocessor instruction sets are classified into groups according to the addressing modes. To get the right addresses for the operands of the current executing instruction, its opcode is disassembled, decoded and then its operand's addresses are taken from registers or memory directly or indirectly depending on the instruction mode.

```
while (the program is not end)
{
        get_opcode_address;
        get_oprd_address;
        execute_the_instruction;
        if(the addresses are not in main memory)
                fault(the addresses);
}
```

Fig.5.2 The operation of the *executor*

To determine if the reference of the current instruction is a valid or invalid memory access, the addresses are compared to the address space in the s_main memory. If it was a valid reference, but we have not yet brought in the page into s_main memory, then a page fault signal will occur. Namely, if the address is not found in the s_ main memory, the executor asks the page fault manager to read the page from s_secondary memory. The further processing for page reading will be explained in the following subsection. Address invalidity is detected by the executor and a demand for the page is transferred to the page fault manager. Figure 5.2 illustrates the operation of the executor. The loop executes the body until a user program is finished. The body consists of tracing control flow by getting the program counter by "get_opcode_address" and operand address evaluation is performed by "get_oprd_address". Then the executor allows the CPU to single step the instruction and stop waiting until the next loop. While the CPU pauses, the executor analyzes the evaluated addresses of the instruction to see if they are valid references or not. If they are invalid in the main memory then it calls the page fault manager to read in the page.

5.2.2 Primary Memory (s_main)

The hypothetical hierarchical memory model in this simulation shows that s_main is a higher layer memory and s_secondary is a lower layer memory. The higher layer memory can be a cache or main memory and the lower memory may be a RAM based local memory, distributed memory or a conventional disk based nonvolatile memory system. In any of these cases, the model is page based virtual memory system in this thesis.

A virtual memory system based on the accurate prefetching of pages operates between the two memories in the hierarchy: s_main memory and the s_secondary memory. Thus, the logical address space of s_main seen by the paging scheme is larger than the physical address space of the s_main. The s_main memory provides a working environment which specifies an address space for a process executed by the executor and it can be changed according to the new pages read into the s_main by the prefetching algorithm. This paging operation has no connection with the UNIX virtual memory system. In other words, the real virtual memory system which is an independent of the simulation is managed by the UNIX kernel. Thus, a real process image provides an execution environment for the UNIX process and it is expected to be controlled instruction by instruction by the executor in Figure 5.2. The valid address space for the executor just depends on s_main memory rather than UNIX memory address space.

The structure of s_main memory is a list of pages which are linked with forward and backward pointers to the next or previous page so that the sequence of pages can be changed easily. Besides the pointers, the s_main memory consists of entries about a start and an end address of the page, a flag showing the status of the page (whether it is occupied or not) and a reference count of the page for LRU replacement scheme. The size of the page is decided by the difference between the specified start and end address of each page. Again, nothing stores any process image in the page because the s_main memory is not used by any instructions or data but is used to provide the address range for

```
struct page page[NUMPAGES]

struct page corememory;
struct page pgfreelist;
```

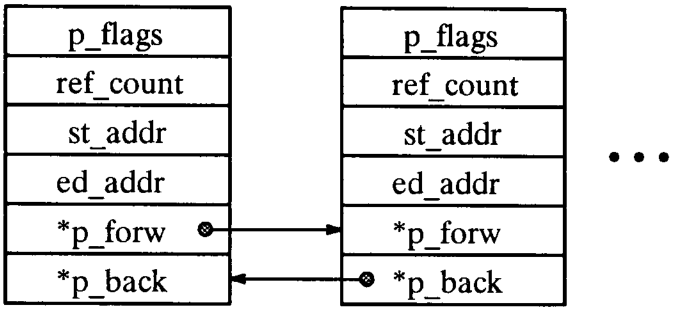


Fig. 5.3 The structure of *s_main* memory

the virtual memory simulator. Figure 5.3 shows the structure of s_main memory. The total size of s_main memory can be varied if the array size is changed. Notice that the s_main data structure looks more like a page table than a memory because it contains most of the control flags for the page rather than the image of a process.

The operations of s_main memory are mainly queue manipulations. If a page is chosen to be read into s_main by the fault manager, the page is allocated from the page free list (pgfreelist) and it is added to the tail of the corememory list after setting the entries in the page. When the page starts to be referenced, then the reference number increases whenever the page is referenced while it stays in s_main memory. If a free page is not available from the pgfreelist, i.e. the s_main is full, a page in the corememory list must be freed and linked back to the pgfreelist and then it will reallocated for the new requirement. When the page is purged from s_main memory, the entries in the page need neither to be saved nor linked into the s_secondary list.

5.2.3 Secondary Memory (s_secondary)

In the hypothetical hierarchical memory model, s_secondary is a lower layer memory such as a disk based nonvolatile memory or remote file server. However, because this s_secondary memory is not assumed to a dedicated disk based secondary memory only, disk seek time in the disk system or communication delay in the remote memory access system are not considered here. The s_secondary memory is viewed as a large capacity virtual address space which is able to contain any size of process address space.

The s_secondary memory does not exist as any special data structure in the simulation. It is a free list linking a set of empty pages to represent a dummy bulk memory. This is also possible because s_main memory does not need to contain any real process image. As illustrated in Figure 5.3, if a page is linked to the **corememory** the page is considered to have been read into s_main memory, otherwise, if it is freed and linked to the **pgfreelist** the page is considered as purged out to the s_secondary memory. In the latter case the page should be flushed of all entries without saving them because the role of s_secondary is only a conceptual in this simulation. However, a real file which contains an executable process's image is stored in a real disk based secondary memory. This provides a necessary working environment for the CPU supported by the UNIX virtual memory system for real paging operation.

5.2.4 Prefetch Queue Management

The prefetch queue management is the most important manager in the AP simulation and it is a unique function compared to other virtual memory systems. In conventional virtual memory system such as OBL, the most obvious problem that arises comes about due to the fact that there is no accurate prefetching management to suppress memory pollution. As was pointed out in Section 5.1, the model of virtual memory

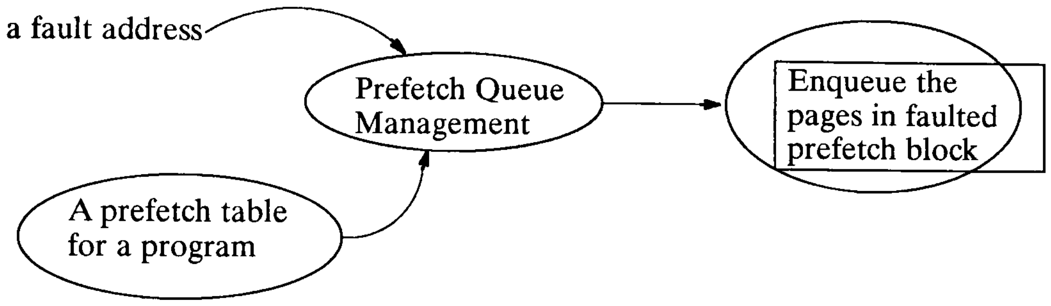

a fault address
Prefetch Queue Management
A prefetch table for a program
Enqueue the pages in faulted prefetch block

Fig. 5.4 The operation of prefetch queue manager

systems was extended to allow accurate prefetching. This comes into existence due to the operation of prefetch queue management and subsequent page prefetching.

The prefetch queue management consists of two major functions: prefetch queue management and lookup of a prefetch block associated with a faulted address in the prefetch table. The prefetch queue is another linked list similar to the memory list which is shown in Figure 5.3. The reason why the two queues have similar structure is that both of them share the same information about the entries in a page as their elements. Thus, if there is a requirement to push a page on the prefetch queue, a page is allocated from the **preqfreelist**. Then the entries – start and end addresses, and the status of the page – of the page are set and it is added to the prequeue list (preqlist). The prefetch queue can have more than one prefetchable page in the list. If a page fault incurs, the page fault manager reads the prefetch queue and prefetches the pages. After finish the prefetching, the pages in the prefetch list are returned to the freelist again. Figure 5.4 illustrates the state diagram of the operation of the prefetch queue manager. The prefetch queue manager reads a prefetch table for a program when the user program starts execution. Whenever a page fault happens, the prefetch queue manager looks for a prefetch block with the faulted address and adds prefetchable pages into the prefetch queue.

```
while (the program is not finished)
{
        get_opcode_address;
        get_oprd_address;
        execute_instruction by instruction;
        if(function_call instruction)
                enqueue argument pages to the prefetch queue
        if(invalid address){
                fault(the invalid address);
                lookup prefetch block for the invalid address and
                enque the pages to the prefetch queue;
        }
}
```

Fig.5.5 The extended executor for AP

The information for prefetching comes from two sources: the prefetch table and object data. The former is a compiler generated table which contains all the relationships described in Chapter 3 and Chapter 4. Suppose we have a prefetch table for a whole program, this table is used for the life time of the process. When a page fault occurs a prefetch block associated with the faulted address has to be specified. To find a specific prefetch block, the prefetch table file is read at the beginning of the simulation and built as a hash table according to the function name. This would add execution time to the process but the overhead of prefetch queue management is relatively small compared to that caused by the page fault. However, it could be reduced if this prefetching part is processed in parallel to the main processor. After making the prefetch table into an internal data structure of the simulator, lookup of a specific prefetch block is straightforward. This operation will be described in detail in Section 5.5. When a page fault happens, only the faulted address is known to the fault manager. This address is used looking for a function and even a precise prefetch block. A prefetch block contains accurate information about future references, so, these are added to the prefetch queue.

The scheme for prefetching object data at runtime will be described in Section 5.6. The information on object data is collected by debugger functions at run time and they will be queued using the same mechanism as described above. The only difference is that the source of information is the prefetch tree but runtime debugger routines called whenever the executor encounters a function call instruction.

Figure 5.5 shows the operation of the executor for prefetch queue management. The bold statements are added so as to be able to prefetch object data pages and the pages in prefetch blocks. A notable point in this figure is that the two operations are independent of each other. The object data prefetching is checked whenever the executor meets a function call instruction and the prefetch block searching is performed only at a page fault because AP is based on a demand prefetching scheme.

5.2.5 Page Fault Manager

Machines whose hardware satisfies the requirement given in Section 2.3 can support a kernel that implements a prepaging system. To implement the algorithms for demand prefetching the hardware must set the reference and modify bits of pages. In this simulation, a software valid bit that indicates whether the page is really valid or not is used.

A crucial issue in conventional operating systems is how to implement a page fault manager in their kernel. In the case of UNIX, the system can incur two types of page faults: validity faults and protection faults but only validity faults are considered in this simulation. Also, UNIX systems can field the fault with the required page in one of five states: on a swap device and not in memory, on the free page list in memory, in an executable file, marked "demand zero" or marked "demand fill". This simulation only considers the pages in an s_secondary memory in the memory hierarchy described in the previous section. The other difference between a real implementation and the simulation is that the modules described in the previous sections – the executor, s_main memory, s_secondary memory and prefetch queue manager – are used to evaluate every aspect of the AP system.

When a process attempts to access a page which is invalid in the s_main memory, it incurs a validity fault and the executor invokes the page fault manager. Address validity is checked by the address evaluator in the code executor and it immediately gives a signal to the page fault manager in order to read in the page containing the address if it is not present. Then the page fault manager reads the faulted page and those pages in the prefetch queue list into the s_main memory. If there are several prefetchable pages in the prefetch queue, the system prefetches all the pages at the same time. While the pages are read into s_main memory, LRU pages are purged out by the replacement algorithm if there are no free spaces available in the s_main memory. Page fault manager consists of four different functions: incore, pagein, pageout and vfault handler. These are described in detail in the followings.

incore(): This routine checks the validity of a current accessing address which is evaluated by the executor in s_main memory. This function is called by the executor to look for every referencing address at pages in s_main memory. Whenever a page is referenced by the executor with a valid address in it, the page is considered to be the most recently referenced. So, the latest referenced page should be linked at the head of s_main memory. Although this routine is quite simple, this is one of the most time consuming process in the simulation because every address for opcode or operand should be checked for their validity and the LRU sequence of the pages in the link is renewed. Each page is set to 512 bytes as a basic size. If the address is found in a page in the list then a positive value is returned. Otherwise, it returns a negative value and then the executor raises an invalid address signal to the page fault manager.

pagein(), pageout(): The basic data structure involved in the page in and out operation is a double linked list. When the simulation system is initialized the freelist pages are established. Whenever there is a demand to read a page into s_main memory from the s_secondary memory pagein() is invoked and it operates that one of the free page is allocated from the freelist and linked to the s_main memory list after updating the page entry. The sequence for reading in a page is that when an invalid address is encountered during execution of a program it calls the vfault() function. The vfault() function allocates a new page in the s_main memory from the free list and sets up the starting and end address of the page and fills some other entities for the page. If there is no space to read into a newpage in the main memory the pageout() function is invoked by the page fault manager. Pageout() selects an LRU page and flushes the entry of the page and links it back into the freelist.

vfault(): This is the main routine that controls the whole demand prefetching operation. It invokes pagein(), pageout() and prefetch() in sequence. Vfault() checks whether the faulted address is in s_main memory as incore() does to prevent a race condition and then if the search is unsuccessful, allocates a newpage from the free list. As stated earlier, if there are no pages available in the freelist, pageout() is called to purge a page from s_main memory. Also, vfault is in charge of prefetching pages in the prefetch queue. If the pages in the prefetch queue are not resident in the s_main memory the pages will be prefetched together at this fault. These prefetchable pages have accumulated in the prefetch queue since the last fault. Then real prefetching of the pages is performed at the next following fault if those pages are still not in s_main memory. The pages in the prefetch queue have waited to be fetched until this page fault occurred. There could be some unnecessary pages to be prefetched for the time between when the pages are enqueued in the prefetch queue and when they are prefetched. To prevent unnecessary prefetching, incore is invoked for each of the pages in the prefetch queue and if they are still not in s_main memory then they are read in.

5.2.6 Page Purging Management

The page purging manager purges out pages that are no longer expected to be used in the near future or that are not recently accessed. The executor wakes up the page purger when a free page is required. The purging operation in this simulation is different from the real implementation in UNIX which has two indexes - low-water-mark and high-water-mark. The Unix page purger is woken up when the available free memory in the system is below the low-water-mark, and the page purger swaps out pages until the available free memory in the system exceeds the high-water-mark. By swapping out pages until the number of free pages exceeds the high-water-mark, it takes longer until the number of free pages again drops below the low-water-mark, so the page purging manager does not run as often. This is quite an efficient scheme.

The page purging manager in this simulation, however, is called whenever page space is required in s_main memory. LRU is used as the fixed space page replacement scheme in the current implementation, therefore, there are no high or low water mark indexes. The doubly linked free page list and s_main memory make it easy to implement LRU. Whenever there is a reference to any pages in s_main memory the order of the pages are changed, thus, the last referenced page is at the head of the list. If the page purging manager is called, the page at the tail of the queue will be purged out. This approach takes most of the simulation time. Some machines set a reference bit when they reference a page, but software methods can be substituted if the hardware does not have this feature as in this simulation. In the clock replacement algorithm which is used in UNIX, the number of examinations by the page purger between memory references can be recorded in the page list but this is not adopted so as to test the effect of LRU in this simulation.

5.3 Generating Prefetch Blocks

The prefetch block is a general notation for a program structure and it provides basic reference information for AP. The prefetch block has a tree structure as an intermediate phase but it ultimately has a table structure so as to be used to predict page accesses. It should be concise enough to represent the control flow of a C+ + program and it should contain sufficient information to predict page accesses accurately. The prefetch block contains some information on function calls, object data and their virtual addresses which are in the same sequence as it's original program. The language that will be used to develop the examples in this thesis is C+ + [Stroustrup 86] and the properties of the language have already been described in chapter three.

The establishment of prefetch blocks is divided into four modules. The first step is to analyze the control flow of a program and then build a preliminary tree using an intermediate language. The second step is to collect information on encapsulated objects and object hierarchies and save them in a tree. The third is to aggregate these separate trees into a single prefetch tree. Since the addresses will be ascertained after linking several relocatable modules of a program in files and libraries, the prefetch tree does not have addresses for symbols up to this phase. Finally, therefore, a process should be carried out to link all prefetch block files for a user program and libraries and to collect the symbol addresses from an executable file. This procedure establishes a complete prefetch block. Each step is described in detail in the following sections.

5.3.1 Generate a Prefetching Tree

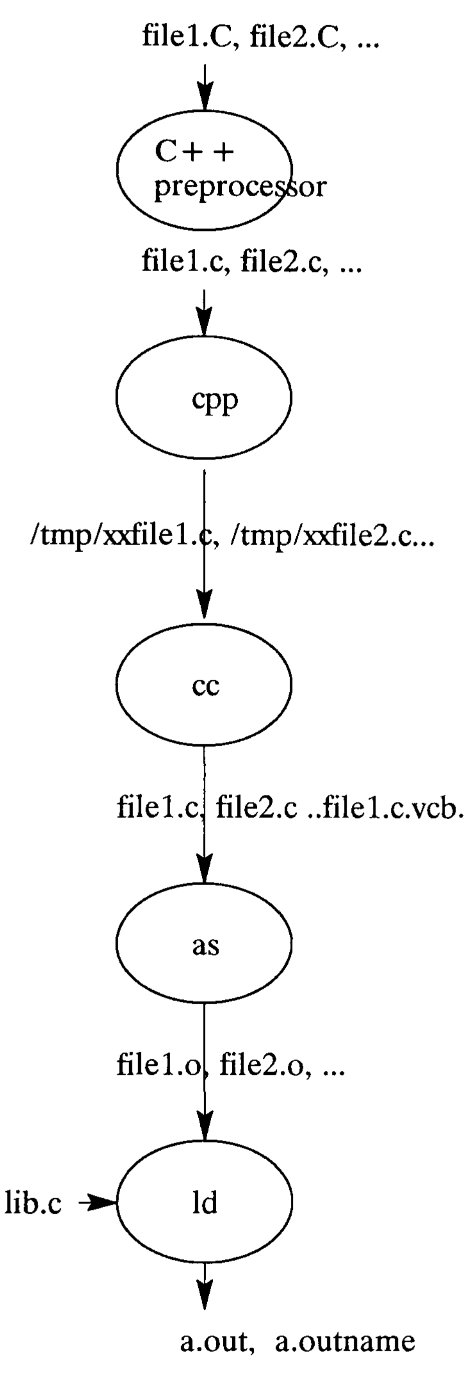
Before the prefetch block is described, the process of how a C+ + program may be compiled must be described. Most UNIX systems support two kinds of object files. First, compilation systems generate relocatable object files, and second, link editors combine relocatable files to create executable files. One can run the program in an executable file because it is a complete image. On the other hand, relocatable files are partial images and typically are not suitable for execution. Linking refers to at least three separate concepts: combining object files, resolving symbolic references, and relocating code so that it may run at particular addresses. Linking is performed by the UNIX utility *ld*. Loading is the act of bringing a program into the address space of a process so that the program any be executed. Loading is performed by the system call exec(). The process of compilation is illustrated in Figure 5.6.

To generate a control flow tree of a program, the GNU C++ compiler, in particular, its intermediate code generation routine, has been extended. The modified compiler supports analysis in terms of control flow and object hierarchy and then synthesis these into the prefetching tree. The process of building the tree is in two steps. A set of instructions make a prefetch block and a collection of prefetch blocks makes a global prefetch tree.

Intermediate codes in a compiler have significant roles such as optimizations. The AP makes use of the intermediate codes for building the prefetch tree. The intermediate code known as a register translation language (RTL) shown in Figure 5.7 is generated when a program is parsed but before it is optimized. The parsing pass of the compiler reads an entire text of a function definition and then constructs partial syntax trees. C++ object and data analysis is also done in this pass, and every tree node that represents an expression has a data type attached. Variables are represented as declaration nodes. Each statement is read in as a syntax tree and then converted to RTL. The RTL generation part is the conversion of the syntax tree into RTL code which is closer in form to assembly language than to the source text. It is actually done statement by statement during parsing but for most purposes it can be thought of as a separate pass[Tieman 89, Pyster 88]. In the GNU C++ compiler, the parse tree is not generated with a function basis but to statements or declarations. Whenever a statement or a declaration is parsed, this parse tree is translated to a corresponding RTL intermediate language list. So, building the prefetch tree can start from this RTL.

RTL has much information which is unnecessary for building a concise program structure. Only necessary information is written into a file (.pb) and this become the frame of the prefetch tree. To establish a prefetching block, the prefetching tree generator reads the RTL code for each function and generates a control flow graph. This is the same process as a code generator that inputs RTL code and generates optimized assembly code. So, this activity belongs to the code generation pass of a compiler. The sequential block in the control flow graph is represented by a sequence of RTL instruction numbers for non–branch or non–call instructions. However, if the prefetching tree generator encounters a branch or a call instruction, the algorithm building a basic block which was discussed in Section 4.1.3 is applied to it. Thus, if the prefetching tree generator encounters a branch instruction, the instruction become a leader of a prefetch block. A label is given for the leader for later reference.

Figure 5.8 illustrates the contents of a .pb(prefetch block) file. A .pb file contains all the instruction sequence numbers for functions in a relocatable file of a program. For example, the function "point_PSpoint" has twenty one RTL instructions and a *primary prefetch block* and a *prefetch block* with the function call. The *primary prefetch block* has a starting label PBB (prefetch block begin) and ending label PBE (prefetch block end) with the block nesting index number. In these prefetch block, both of the index numbers are "1" because they are not nested in each other. If a block is nested in another block, the nested block has the index number n + 1. Also, PBB has a RTL instruction number for the end of a block. For example, for the first PBB, JD = 12 thus, RTL 12 is the end of the prefetch block. In addition, prefetchability such as a "*primary prefetch block*(ppb)" or "*prefetch block*(pb)" of the prefetch block is recorded at PBE. The prefetchability of the prefetch block will be used to aggregate the blocks to a bigger prefetch block later on.



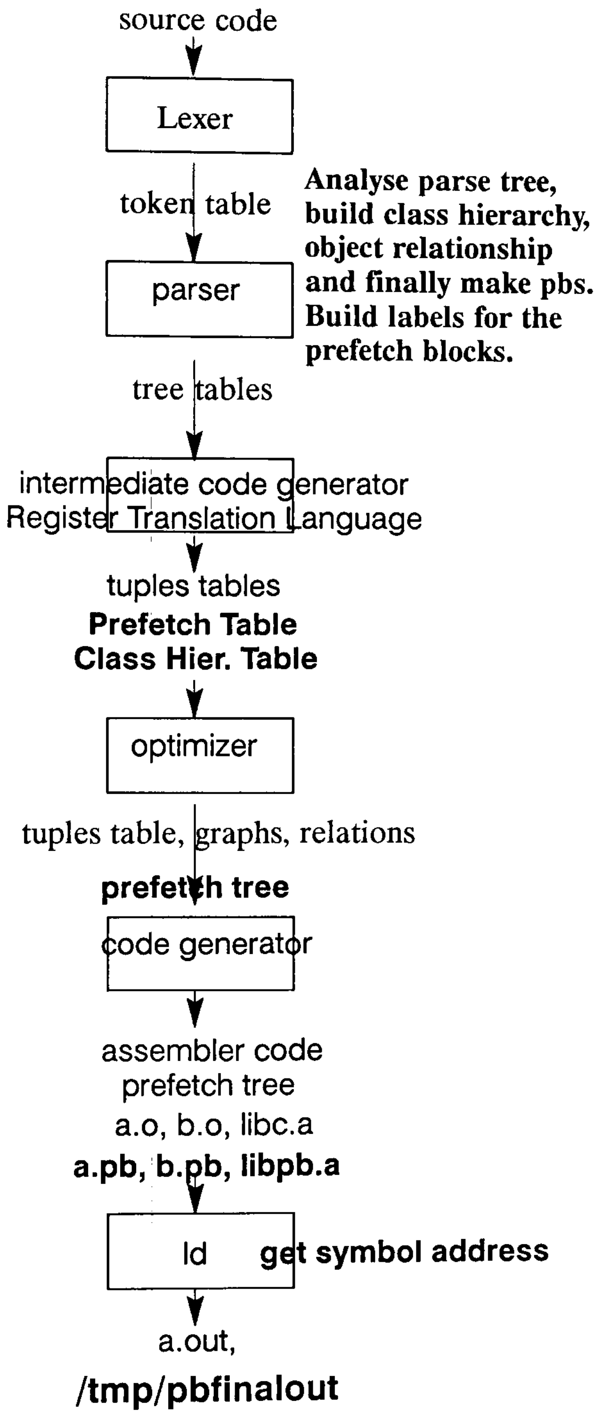


Fig. 5.6 Global structure of the compiler

Fig. 5.7 The structure of g++ for AP

Also, if the prefetch block generator runs into a function call instruction which is important information in the prefetch block, the invoked function name is recorded after the RTL instruction sequence number. This name will be used to find the function's address from the symbol table. The prefetch block tag numbers will be used to manage nested blocks properly. If there are backward branches they should be sorted out at the end of the processing for a function, so, the prefetch block generator must have a two pass look up to deal them.

```
;; Function point_PSpoint

 2  3  4  6   7 PBB 1, JD=12
 8  9  IFT __builtin_new
10  11  12  PBE 1  pb
 5  13  14  15  16  17 PBB 1, JD=21
18  19  20  21  PBE 1  ppb

nonprefetchable
```

Fig.5.8 A .PB File

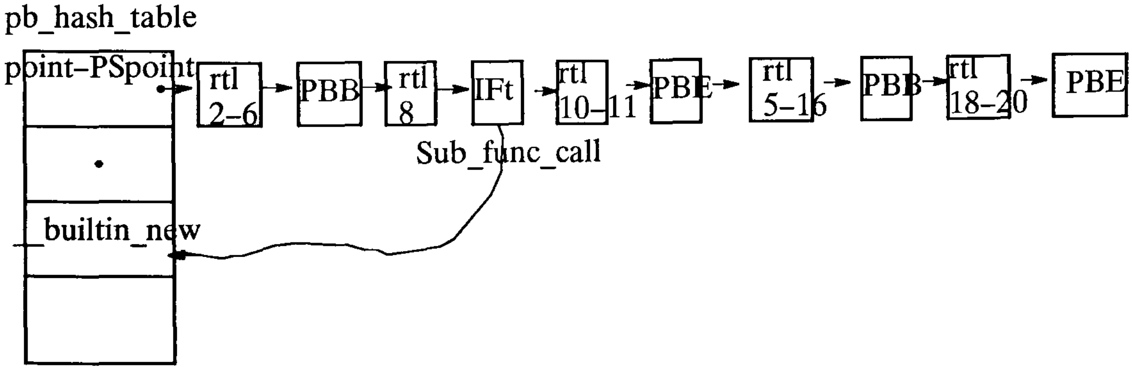


Fig. 5.9 The primitive program skeleton in Fig.5.8

To build the prefetch tree, the .pb file is re-read by the prefetch tree generator so that the instruction number, PBB and PBE become entities in the tree. The prefetch tree can be represented by a variety of data structures but a tree structure was adopted in this simulation. This is because a tree structure can represent the flow of control best. A hash function table is built for a quick search for a function in a program and each entity in the table provides the head of the tree of a function. Prefetch blocks, function calls, position labels then become branches in the tree. For example, Figure 5.9 shows a prefetch tree for the function "point_PSpoint" in Figure 5.8. An entity in the hash table is allocated for the user function and it also has the first PBB from RTL instruction 2 to 6. Then, a node is built for PBB, IFT and PBE. The IFT node has a pointer to the function in the hash table. If it is not already in the hash table in the first pass, the linking is postponed to the second pass of searching. The linked list data structure is useful to optimize the prefetch block and deal with backward branches. After completely establishing the prefetch block, it is saved in a file which is named by appending .pb to the source file name. When the basic process of building the prefetch block is completed the prefetch block generator starts to aggregate some sequential prefetch blocks to make as big blocks as possible. This is described in the following section.

Figure 5.9 illustrates the data structure of the prefetch tree after completing the linking of prefetch blocks for functions in a program. The hybrid data structures are a hash table for each function, a linked list and a tree which represents the prefetch blocks in each function. Again, each prefetch block can be represented by a list of records consisting of an identifier, a sequence of function calling and branches in the block, followed by a pointer to the leader of the block, and by the lists of predecessors and successors of the block. The hash table is called "pb_hash_table" and it enables fast searching for function names. This tree will be used by relookup() for linking function calls and labelling each prefetch block.

A problem with implementing the prefetching tree in C+ + is that the GNU compiler generates assembly code on a function basis. And prefetching encompasses all the functions in a program. To keep the structure of the present compiler, the prefetch block tree is established independently from the RTL and assembly code generation pass. It is important to note that an edge of the flow graph from block B to block B' does not specify the conditions under which control flows from B to B'. That is, the edge does not tell whether the conditional jump at the end of B (if there is a conditional jump there) goes to the leader of B' when the condition is satisfied or when the condition is not satisfied. That information can be recovered when needed from the jump statement in B.

5.3.2 Merging Prefetch Blocks

By now, a program skeleton has been made in a tree structure representing the control flow of a program. A hash table contains all the function heads in the program and each function head become a root of the control flow tree for the function. Some important information about a function such as a start block, end block and sub_function calling become a node of the program. In establishing the prefetch blocks in the previous section, we realized that some primary PB are contiguous. For example, RTL instructions 5, 13 to 16 in Figure 5.9 are sequential to the following PBB. Therefore, merging and optimization for some primary prefetch blocks (sequential) and some prefetch blocks in the prefetch tree should be performed. Consecutive primary prefetch blocks or prefetch blocks are combined into a prefetch block. For example, suppose a prefetch block has a function call then the primary prefetch block before the function call statement and the first prefetch block in the invoked function can be merged into a bigger prefetch block. The benefit of this merging strategy is that we can prefetch more and more references accurately. This merging operations should be carried out for a whole program and, therefore, a global control flow graph is required.

How should be represent a global control flow graph for a whole program ? The parse tree in the compiler is built for each statement rather than for a whole program. Most conventional compilers do not generate any information about inter-function relationship because it is not necessary in a stack based machine. However, the prefetching compiler must be able to generate some inter–function control flow graphs using symbols as discussed in Chapter 4, i.e. a sequential block or function calls which would be able to replace it in a subgraph. The methodology used to achieve this is to build a prefetch tree having nodes for each prefetch block and function call to link caller and callee function to each other by pointing to the callee function at prefetch tree generation time.

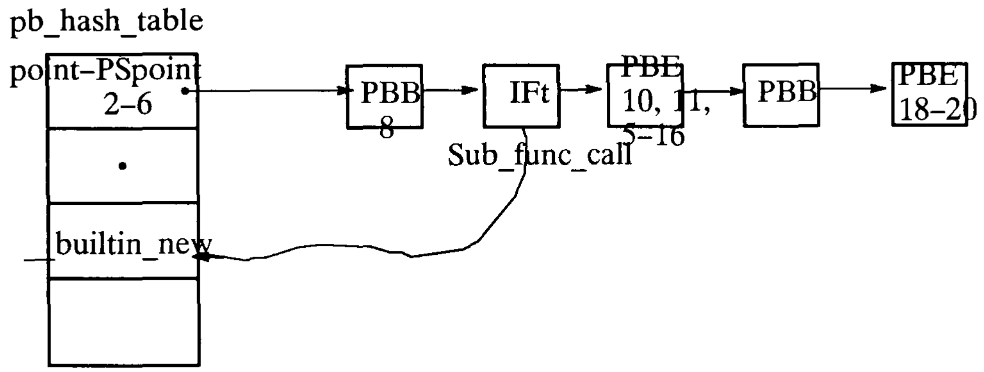


Fig. 5.10 An optimized program skeleton

Figure 5.10 illustrates the tree in Figure 5.9 after merging sequential blocks into their preceding prefetch blocks. The difference in the two trees is that the sequential block (RTL instruction 2 to 6) is absorbed into the function head and the second sequential block (RTL 5 to 17) is also merged into the PBE. As shown in this figure, a sequential block is merged to the previous prefetch block and eventually the PBE of the previous prefetch block is extended to the end of a merged sequential block end. Merging inter–function blocks is performed at the very last stage of building the prefetch tree and will be described in Section 5.3.5.

Two pass searching is necessary to complete the linking between a caller and a callee function. In the control flow graph some functions which are not able to be looked for at the first pass of generating the program frame are looked for again here after completing a preliminary linked list. If the functions are found then they are chained to the global prefetch tree.

5.3.3 Building Class Hierarchy Trees

In C+ +, some objects are like other objects in an object hierarchy. These objects in the same hierarchy have several implications. As stated earlier in Chapter 3, the *yoyo* problem, a series of constructor and destructor invocations and dynamic bindings occurs in the hierarchy. As far as accurate prefetching is concerned, yoyo problem can be encompassed by control flow analysis which has already been described in the previous sections because the function invocations involved in yoyo problem are expressed explicitly. However, an explicit object hierarchy is required to adopt static grouping among dynamically bound functions in the hierarchy. Thus, this hierarchy tree is built to enhance the locality of reference by glueing some member functions in the hierarchy together. The tree will also be used to prefetch constructors and destructors in base classes. The constructors and destructors are in a sense explicitly expressed in the functions but it is easier to put them in the class hierarchy tree.

There are no internal data structures which represent the full object hierarchy in the compiler. So, AP's own object hierarchy table is built to make inter–object relationships at compile time. Existing data structures for virtual functions can be used for the object hierarchy but these are not enough because they are not able to represent the whole object hierarchy in a program. For example, Figure 5.11 illustrates the data structures for an instance of a class Point and its virtual function table in the OOPS library[Gol 87]. Each object inherits a pointer, *vptr*, to its virtual function table, *vtbl*. The virtual function table pointer, *vptr*, is inherited from class Object in the library. Both the *vptr* and *vtbl* are managed by the C+ + translator as an internal structure and are not accessible to the user but they can be shown in the preprocessed code compiled by

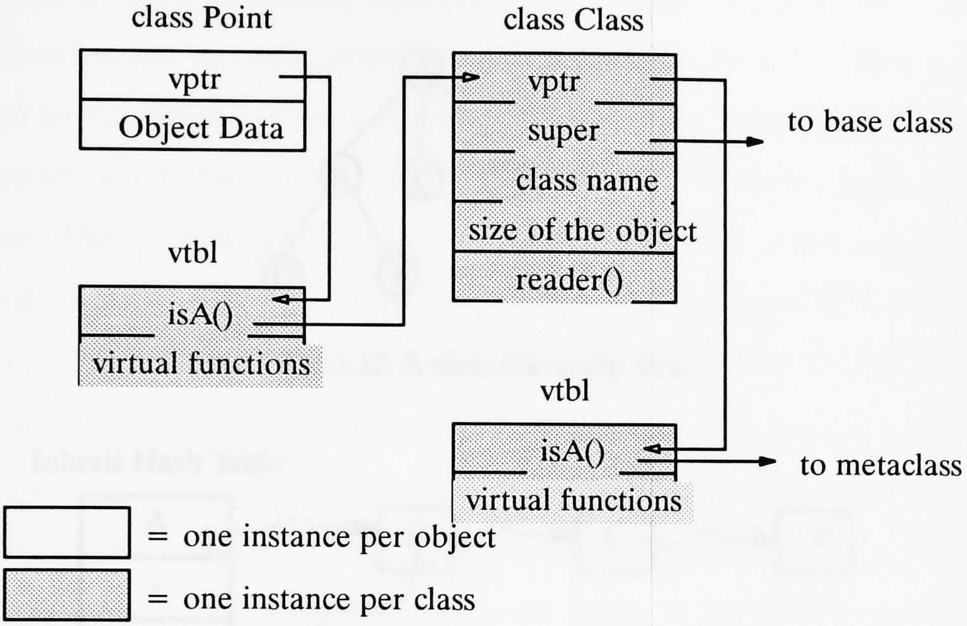


Figure 5.11 A virtual function table in C++

standard C++. This table enables dynamic binding by lookup for a proper implementation of virtual functions. Therefore, the builtin virtual table cannot to be accessed for establishing the inter–object relationship table because it is built for a limited set of objects: those which have virtual member functions.

A tree is, therefore, built for each independent class inheritance hierarchy to group constructors and destructors in its base classes as well as some member functions. The data structure for it is a simple tree with a hash table for easy lookup of object names. The hash table is used to hash base class names in each hierarchy. Figure 5.13 illustrates an AP's class hierarchy structure for the tree in Figure 5.12. Whenever the parser encounters a class with its base class (class D: B) it searches the tree to see whether the object's base class is already present. If it is a new class, the class is managed as a base of the hierarchy by allocating a new entry in the hash list. If its base class is already registered in the tree, the class belongs to existing hierarchy. Siblings in the same class are pointed to

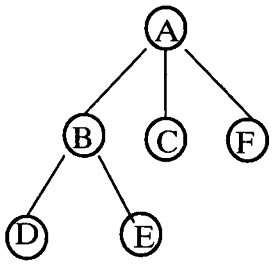


Fig.5.12 A class hierarchy tree

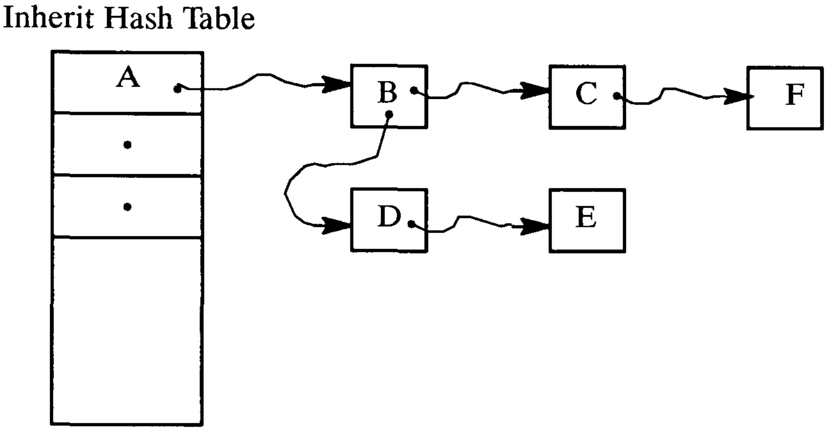


Fig.5.13 The data structure for the tree in Fig.5.12

with a forward pointer and subclasses are linked on beneath it. In current implementation, multiple object hierarchy is not considered.

5.3.4 Relational Table for Object Member Functions and Its Data

Encapsulated operations and data structures in an object are important information for prefetching in the AP. As explained in chapter three, when either of the encapsulated parts is accessed the other part can be prefetched before it is read into main memory by a fault. The implementation is done in two steps. Firstly, an object name is mapped to a class name then the object's data are linked to member functions. The data structure used for building the relationship is basically the same as that in the object hierarchy tree. Every new class's entities in a C+ + program are recorded in a hash table named class_base. For member functions, C+ + classes overload their constituent function names automatically. When a function name is declared in a class, its name is changed to its overloaded name during compilation. For example, since names for constructors and destructors can conflict, a leading '$' is added for destructors in the compiler. When a constructor is encountered during processing, a new entry in the hash table is allocated for the new function. If the function is a constructor, then it should have a pointer to its class name so as to be able to look for its constructor in the base class. The class hierarchy tree which is described in the previous section is a route looking for its base class. Then all the names of member functions which belong to the class can be identified by the compiler to record them under the class name. If a destructor is processed, the auto-delete() function in the C+ + library is added by the compiler.

Figure 5.14 shows the hash table with linked list for the relational table. Whenever a new class is declared, an entry is allocated for the class and it become a head of a list. The list contains member functions as an entry in each node and each node has its base class name.

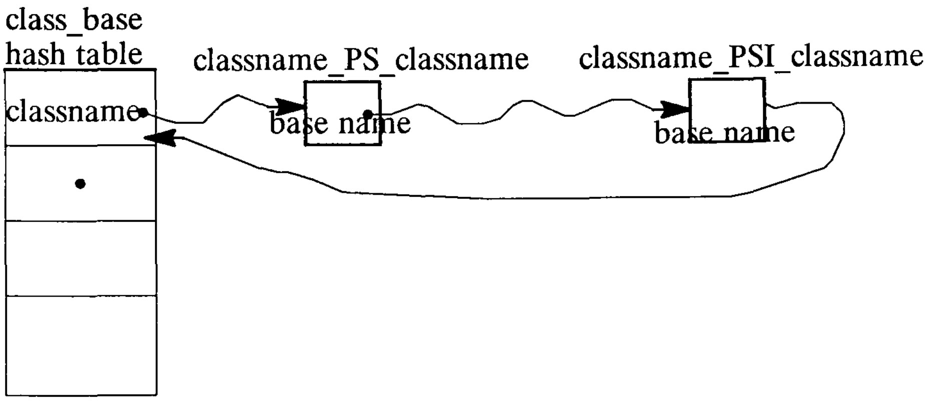
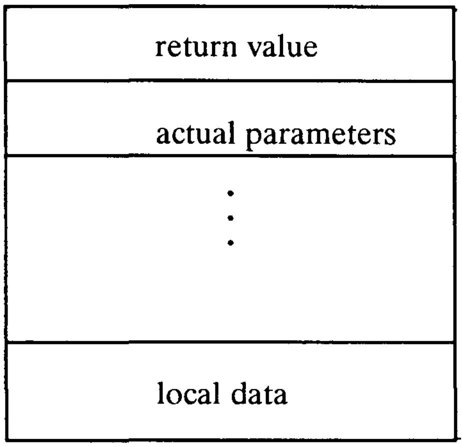


Fig. 5.14 A relational tree for objects

5.3.5 Collecting Object Data and Variables for Functions

The prefetch block provides an accurate representation of the flow of control of the program and contains some information on object references. As for data prefetching in prefetching blocks, the names and locations of data items should be collected and those related to a faulted prefetch block can be prefetched by providing information to the fault manager. So, the object data and variable references are collected at this phase and they are appended to the prefetch block.

Not all variable references need to be prefetched. For instance, references to variables that are either register variables, variables whose addresses are computed at run–time, or local variables that are in the stack are not meaningful in terms of prefetching. The reasons are as follows: first, register variables do not make memory references. Secondly, variables whose addresses are not known during compilation cannot be prefetched using the prefetch tree because the determination of location is delayed until runtime. Implementations of languages like C use extensions of the control stack to manage activations of procedures. Data objects whose life times are contained in that of an activation can be allocated on the stack, along with other information associated with the activation. However, all data objects in, for instance, Fortran can be allocated statically. One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code[Aho 86]. Stack variables do not have fixed addresses that are known when the program is compiled, because, for languages like C, it is common to push the activation record of a procedure on the run–time stack when the procedure is called and to pop the activation record off the stack when control returns to the caller. The structure of a general activation record is shown in Figure 5.15 and local variables in the activation record can be found in the lower second row. However, the stack is not a large component of the miss ratio since it is accessed often with good locality and the referenced parts of the stack do not tend to get replaced. Therefore, stack variables do not need to be prefetched[Brent 87]. Figure 5.16 shows a typical subdivision of run–time storage organization in which a different storage allocation strategy is used for each area. Static data allocation lays out storage for all data objects at compile time. Stack allocation manages the run–time storage as a stack. Dynamic allocation allocates and deallocates storage as needed at run time from a data area known as a heap.



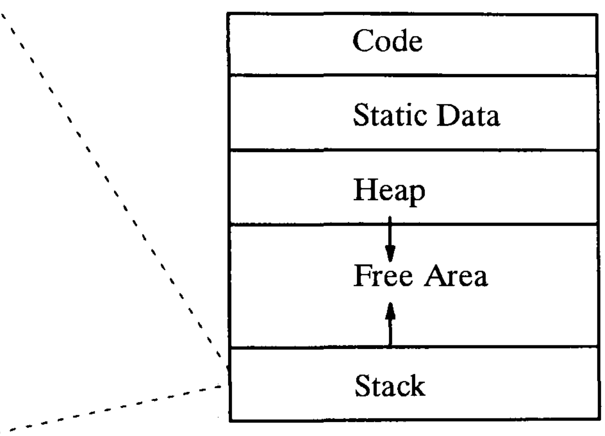


Figure 5.15 A general activation record Fig.5.16 A subdivision of run–time memory

To collect global and static variables as well as object data which are accessed by member functions, the same type of hashed linked list described in the previous section is used. When the compiler parses a new function an entry in the hash table is allocated for it. Then, for all new variables or object declarators encountered during the parsing, new nodes are allocated if they are not already queued. A hash table called symbol_tab is used to build a linked list of variables defined in each function. This variable table (symbol_tab) is tied to the prefetch table which is discussed in Section 5.2.2. A member function has an entry in the prefetch table and the symbol_tab. The pointer sub2_func_call in the prefetch table links the entities together by pointing to the function in the symbol table. Figure 5.17 shows the combination of prefetch tree and the variabls tree. Addresses of these object and variable will be filled in by the function symbol_address() which is described in the following.

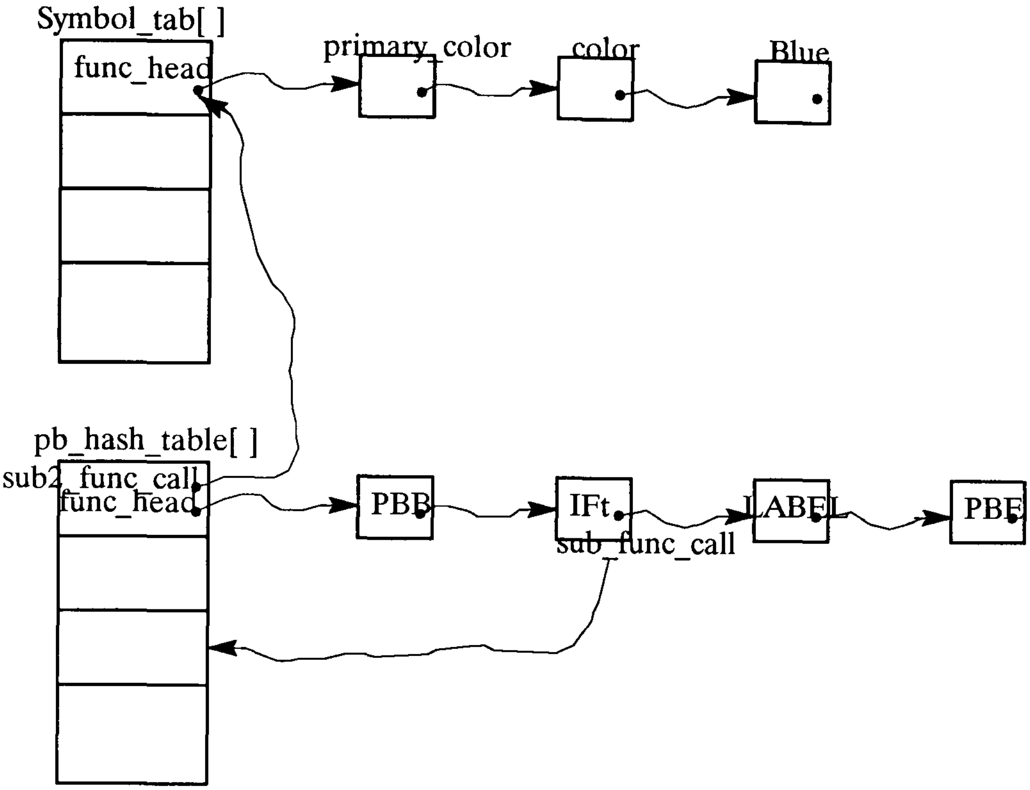


Fig. 5.17. An Intermediate Prefetchtree for a Function

Most object data collected in the tree are generated by using symbol generation routines in the GNU C+ + compiler provided for symbolic debuggers. The compiler was extended to generate variable names for the symbol table which we need only for prefetching. Notice that this scheme for managing object data and variable for prefetching is almost the same as Brent's work. However, AP has an additional prefetching scheme at runtime to cover some dynamically created object data and variable prefetching. This is discussed in Section 5.5.4 in detail.

This way of building a prototype is acceptable because some existing routines in the compiler can be extended for prefetching. The operational command to generate the symbol table for prefetching is "g+ + -V -dv filename.cc". When an example C+ +

program is compiled with the -V option, it gives hint to the code generator to generate only the necessary symbols for AP.

5.3.6 Combine All the Trees into a Prefetch Tree

By now, several trees have been built to represent a program frame based on control flow analysis, class hierarchy tree, encapsulation trees tying encapsulated member functions and data, and a linked list that collects variables related to functions. These independent trees must be combined into a prefetch tree to contain all the information described for prefetching. Otherwise, the prefetch queue manager would have to search every tree at every page faults. Since all the trees have same data structure, it is easy to link them together into a single tree.

Each tree has a hash table as a root for each function and these hash tables are identical because the keys of the hash tables are the function names. The class hierarchy tree has different entities in the hash table where nodes in the tree have constructors and destructors. Those function names can be linked when they are referenced. Therefore, the final prefetch tree is established by linking all the tables by using pointers. This final prefetch tree cannot be kept as an internal tree of the compiler anymore because the necessary address for the symbols in the tree will be specified by a linker which is a totally different phase. This is similar to the way that all assembled files are written in relocatable files and kept until they are linked. So, the final prefetch tree is saved in files named by the source file name with .pb_in appended. The number of .pb_in files is the same as that of source files.

The object hierarchy tree in Section 5.2.3 was built to enhance the locality of reference by combining some member functions in the hierarchy together. The tree is also used for prefetching calls on constructors and destructors in ancestor classes. The constructors and destructors are in a sense explicitly expressed in the function but it is easier to call them if we use the object hierarchy tree. The use of this hierarchy tree is that it should be combined with the prefetch tree because when a constructor in a leaf class is called, all the constructors in the ancestors should be listed in the prefetch tree.

When the prefetch generator encounters an object declaration the class name of the object is searched for in the inherited hash table. The class name is used to look for its ancestor classes as well as their constructors in the hashing table of the object hierarchy tree. If the class name is identified the names of the constructor and destructor for the class are copied to the prefetch tree. This operation goes on until all base classes are sorted out. Moreover, the relational table for object member functions and their data can be used to combine the prefetch tree and the objects. Consequently, the prefetch tree contains most information for prefetching objects. However, the address of objects are not yet specified.

5.4 Getting the Address of Objects

Addresses of objects are the most important information for an accurate prefetching scheme. The prefetching tree has most names to be prefetched but the location of the objects are not yet known because it is feasible to leave the relative positions of the activation records unspecified and allow the link editor to link objects, executable code, object data and activation records.

The addresses of member functions and static variables are fixed at linking time of the program. Most compilers generates all symbol names at compile time but the relocation addresses are unspecified because relocation is not performed until linking all library functions and user programs in separate files. The compiler operates on just one file at a time and cannot correlate a declaration in one file with a reference in another file. The linking loader, *ld*, does just this. It takes a collection of object files and builds one executable binary file by resolving all external references. It scans all of the object files being linked for the declaration of an as yet unresolved identifier. *ld* can be told to search any number of previously developed libraries for declarations as well.

One simple method to get the address of symbols from a final executable file is extending the namelist program(nm). The namelist (symbol table) program prints out global and local symbols from an executable file with an appropriate format. So, the addresses generated for every object by the namelist program can be filled into the entities in the prefetch tree. The addresses of objects and variables which are left blank in the prefetch tree at compile time would be filled in at this stage by the symbol_add() routine. Symbol_add() looks up all addresses for objects, builtin functions and static variables from the symbol table in an executable file and fills them in the prefetch tree so as to complete the tree. However, this method takes a long time, proportional to the number of symbols, as looking for every symbol needs a search of all the symbols in a file.

The alternative is to search an internal symbol list in the linker. Every symbol is reprocessed by the linker to check undefined or multidefined symbols at link time. So, they can be read into the symbol list in the linker from reallocatable files. The symbols are grouped together into two categories. One is global symbols which are saved in an internal data structure, a linked list, and the other is local symbols which are not of much interest to this implementation. The global symbols have addresses but the local symbols have relative addresses. So, finding symbol addresses for the global symbol list can be done by searching the internal symbol list in the linker.

The further processing covers combining all prefetch files, which were established by the compiler/assembler and saved in files named .pb_in, for a user program and some library files which will be included in the source files, into a final prefetch file named the *pbfinalout*. Figure 5.18 illustrates linking of prefetching files for library functions and a user program into the final prefetch table. The prefetch files for a library are already built, archived and stored in the same directory as the library files. The archived files contain all the prefetching information for library source files and these are archived like an ordinary library. Furthermore, addresses for every symbol in the prefetch table files

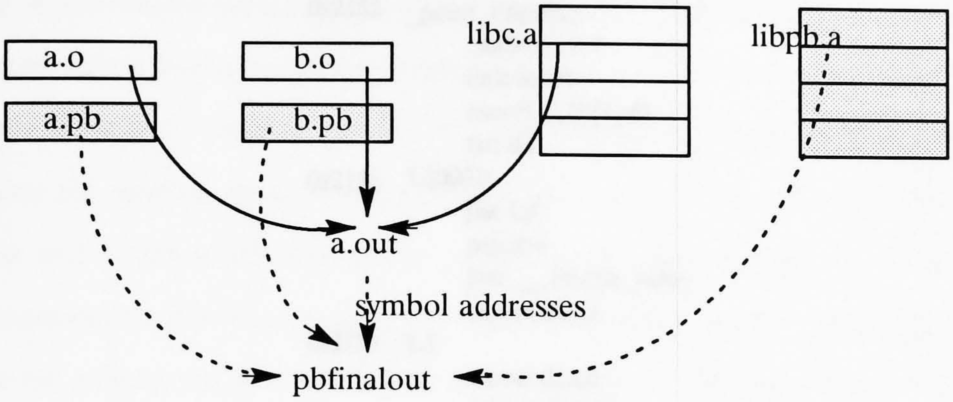


Fig. 5.18. Building a *PBfinalout*

are searched for and taken from the symbol files which are read into the list at the early stage of loading. When a prefetch block file for each text file is processed, global symbols in the prefetch table are sought in the global symbol lists. Searching for symbol addresses for the global symbol by this method takes less time than the previous method by symbol_address because it makes use of an internal symbol list which is built by the linker.

5.5 Naming of Prefetch Blocks

In the AP system, when a process encounters a page fault the prefetch manager looks for the relevant prefetch block which the fault refers. The prefetch block contains information such as flow of control, encapsulated objects, object hierarchy prefetchable object functions and variables with addresses. To identify the prefetching block which is uniquely defined in the *pbfinalout*, a naming system is necessary to address the right prefetching block which is associated with the fault. A naming scheme for the prefetch blocks is discussed in this section.

To address every prefetch block, labels are added to each prefetch block by inserting dummies during the intermediate code (RTL) generation phase in the compiler.

```
0x2152  _point_PSpoint:
              .stabd 68,0,4
              link a6,#0
              movel a6@(8),d1
              tstl d1
0x215c  L10001:
              jne L5
              pea 8:w
              jbsr ___builtin_new
              movel d0,d1
0x216a  L5:
              movel d1,d0
              movel d0,d0
0x216e  L10002:
              jra L4
              .stabd 68,0,4
0x2170  L4:
              unlk a6
  XX          rts
```

Fig. 5.19 Pseudo labels in a prefetch table

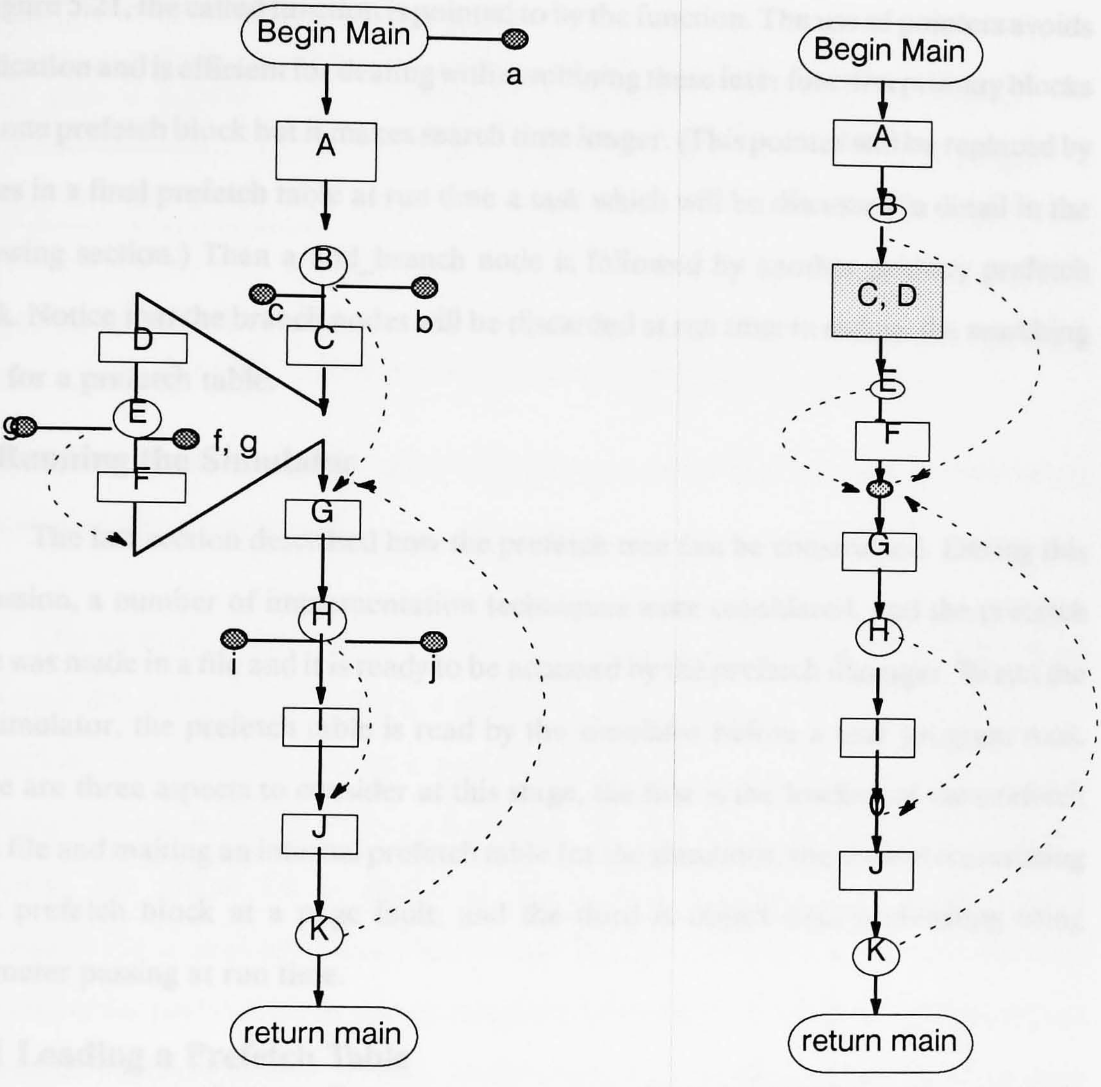
These labels have the form L10xxx for branch points and L20xxx for function call statements and they are unique within the file in which they are defined. Then, the way of getting addresses for each prefetch table labels into same as for an ordinary variable, namely, the addresses for the labels can be taken from an executable file at linking time as with other labels. For example, Figure 5.19 shows two dummy labels for the two prefetching blocks. The first one is an ordinary label for a function name. The second label L10001 is inserted to identify the prefetch block starting with "jne L5" and it ends with "movel d0, d0". The third label L5 is an ordinary label generated for internal jump. The fourth is another prefetching label just before a branch instruction. The figure shows that the prefetching labels have addresses like ordinary labels.

A RTL label instruction is inserted just before a jump instruction. This is done by *final()* when all optimization is finished. After compilation, the intermediate labels will be saved in the resulting symbol table in a relocatable file by giving the –L option to the SUN assembler. Without the option, all the defined labels are discarded in the assembling stage. A noteworthy point to make here about adding labels is that each .pb file may have the same labels, in particular, those labels for prefetching, which are defined uniquely in the file. However, these labels are not ambiguous when searching for their address because the symbols must be uniquely defined as local to the file. Symbols for branch points and function call statements belong to this class of symbols. This method of implementation provides fast and efficient look up of for symbols' addresses. How do we know the address range of the last label in a function ? The last address of a function is required to specify the address range of the function. This can be implemented by generating another label XX just before a return instruction, rts. In this example, the label just before the end of unlink, unlk, is used to generate the last address of the function. The last address of the function _point_PSpoint is L4 + 6 in Figure 5.19. Now, every prefetch block has a unique address and this provides a name for a suitable prefetching block at a page fault.

To achieve accurate prefetching, an arbitrary faulted address which belongs to a function should map to a prefetch block because it is uniquely defined in the program. Since the prefetching tree was built as a skeleton of the program, all function names in the program can be found in the prefetch tree as well. However, addresses for data are not defined but they can be prefetched if they are related to the functions. For function calls in the prefetch tree, only the caller contains prefetching information with regard to its called functions by pointing to their prefetch blocks. For example, if a prefetch block *A* has a function call to *fred()*, prefetch block *A* will have a pointer to the prefetch block head of *fred()*. This method avoids multiple descriptions for function invocations and makes for a simple prefetch tree. However, the prefetch block of the caller function will contain all the prefetch information including for the callee without pointer when the prefetch tree is read and saved at an internal data structure by the simulator. This is described later in this section. For object data and variables which are statically defined at compile time, the head of the function block contains the necessary information.

To illustrate how a prefetch table is generated and maintained for accurate prefetching, a modified form of the prefetching tree diagram used in Figure 5.20 is employed. Consider an application program containing two functions, one main() and the other fred() which is invoked by main(). In Figure 5.20, the prefetching diagram can be translated to a simpler diagram which is optimized from the original flow graph. The optimization is particularly done for function calls so that a function call symbol can be replaced by an equivalent subblock. For instance, the first primary prefetching block, D, in fred() is combined with the last primary prefetching block, C, in the caller function. This combined subgraph is shown as a shaded box in Figure 5.20. Also, Figure 5.20 has branch destination points (shaded circles) for notational convenience. Now, this diagram consists of primary prefetching blocks and arcs from sources to destinations. The arcs have hardly any meaning in terms of prefetching because they bypass control flows. As stated earlier, branch points have labels of the form L10xxx and function calls L20xxx. The labels can also be used to distinguish prefetchability for a primary prefetching block or a prefetching block. If a block is a primary pefetching block, the page fault manager does not need to search information for prefetching but use an one block lookahead. However, if a block is a prefetch block, this block may have some prefetchable blocks which are related to function calls because this block might be an optimized subblock rather than a simple prefetch block. The fault manager should therefore search the prefetchable informations.

The symbol diagram can be transformed to a linked list with a hash table for the function names. The head of each list contains a begin symbol and a pointer to the first prefetch block. The second node contains a begin_branch which is always coupled with an end_branch. The branch node pair makes a prefetch block. Node C is a primary



Prefetch Tree

Optimized Prefetch Tree for Main

Fig. 5.20 Naming of objects in a prefetch tree

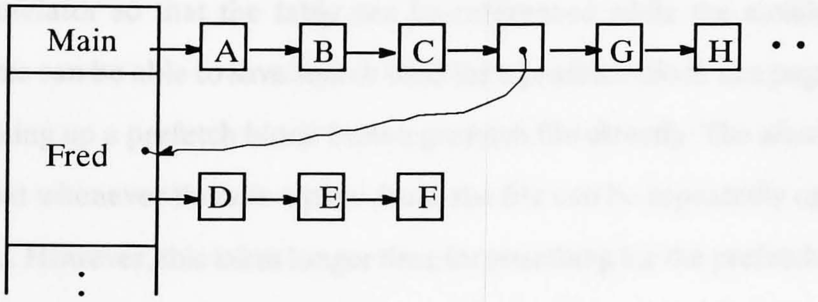


Fig. 5.21 Prefetch tree for Figure 5.20

prefetch block and it points to the next node which is a sub function invocation. As shown in Figure 5.21, the called function is pointed to by the function. The use of pointers avoids duplication and is efficient for dealing with combining these inter function primary blocks into one prefetch block but it makes search time longer. (This pointer will be replaced by values in a final prefetch table at run time a task which will be discussed in detail in the following section.) Then a end_branch node is followed by another primary prefetch block. Notice that the branch nodes will be discarded at run time to reduce the searching time for a prefetch table.

5.6 Running the Simulator

The last section described how the prefetch tree can be constructed. During this discussion, a number of implementation techniques were considered, and the prefetch table was made in a file and it is ready to be accessed by the prefetch manager. To run the AP simulator, the prefetch table is read by the simulator before a user program runs. There are three aspects to consider at this stage, the first is the loading of the prefetch table file and making an internal prefetch table for the simulator, the second is searching for a prefetch block at a page fault, and the third is object data prefetching using parameter passing at run time.

5.6.1 Loading a Prefetch Table

A prefetch table saved in a file named *pbfinalout* is read into an internal data structure in the simulator so that the table can be referenced while the simulator is running. This scheme can be able to save search time for a prefetch block at a page fault compared with looking up a prefetch block from a prefetch file directly. The alternative to this scheme is that whenever there is a page fault, the file can be repeatedly opened, searched and closed. However, this takes longer time for searching for the prefetch block which is associated with a page fault than looking up an internal linked list. Even the table

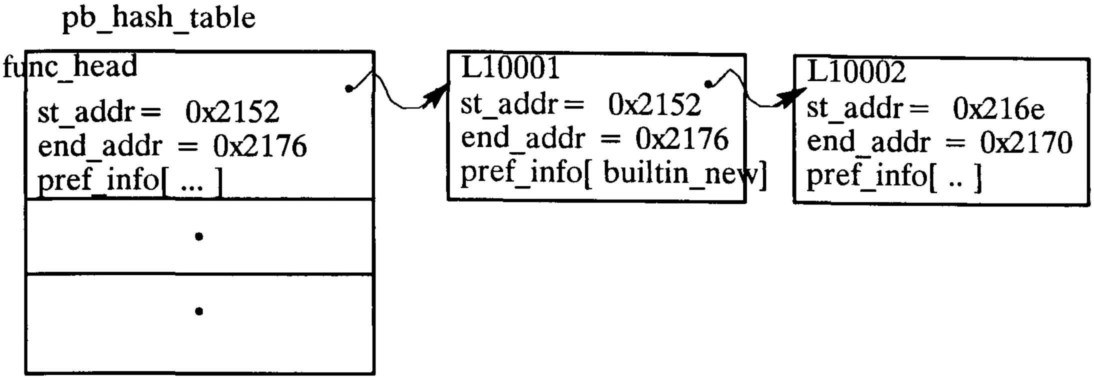


Fig.5.22 A final prefetch table reference by the simulator

searching time is very critical for improving virtual memory performance because this is an operating system overhead.

One method adopted to reduce the searching time in this simulation is shown by the linked list in Figure 5.22 where a pointer is given for a callee function to prevent duplication. However, the pointer is replaced by values for the callee function so that searching time can be saved. This is a final prefetch table which is referenced by the AP simulator. The linked list as shown in Figure 5.22 contains all the prefetchable information for a prefetch block. Each node has four entries: label, start and end addresses of the prefetch block and an array of prefetchable information. The labels can be grouped into three categories: func_head, L10xxx, L20xxx. Firstly, func_head is always in the hash table and it corresponds to the symbol 'begin function'. Func_head labeled nodes contain two addresses which specify the range of the functions. Also, the first prefetch block of the function is included in the head node by the merging scheme described in Section 5.2.2. Secondly, L10xxx labels represent a prefetch block. They contain start and end addresses for the block and some prefetchable information related to static variables. Thirdly, L20xxx labels represent a function invocation in the block. So, this node must have contained some prefetch information in it. The information was pointed to by a pointer in the prefetch tree but when it is read into the prefetch list in the simulator, the references are turned into values. The addresses in the node specify the address range of the block but the end address is in the caller function rather than the end of a callee function.

As it is stated earlier, address of the labels are desirable when looking for a position with the missing address where a fault is incurred. The final prefetch table has address ranges for each prefetch block in the program. Thus, a necessary prefetch block can be found by table look up. The information for a prefetch block can be found with a faulted address and it will be used eventually to prefetch the prefetch block.

5.6.2 Searching for a Prefetch Block at a Fault

How do we look for prefetching block corresponding to a faulted address ? The way of looking for the appropriate prefetch block from the prefetch list in the simulator is to find the function with the faulted address in the hash table. The hash function is mainly used to fill in prefetching information in the table but when searching a PB, address ranges are used. Because the only information available when looking for prefetching block is a faulted address, we look for a faulted function and then for a faulted prefetch block by searching the linked list in the function.

Different labels have different schemes when looking for prefetching blocks. Firstly, the address range in a function_head label specifies from the starting label of the function to the return instruction of the function. Generally, the code area of memory is filled up with consecutive user functions and library functions. So, any addresses in the code area must belong to a function. The function_head labels specifies the address range for any function in the program. Secondly, L10xxx type labels are assigned for every branch points as dummy labels to identify each prefetch block. Before we optimize the prefetch tree, there are a number of small patches of prefetch blocks but some consecutive prefetch blocks are combined into bigger prefetch blocks. So, the addresses of prefetch labels specify the beginning address and the end address of the prefetch block including the merged sequential blocks. Thirdly, L20xxx type labels are used for every function call statement. The labels are unnecessary for naming prefetch blocks but they are used to look for a proper prefetching point in a prefetch block. The usefulness of the L20xxx label is that it provides an address by the function invocation statement, namely, it is used to determine whether it is lower or higher than the faulted address. For example, suppose a prefetch block has two prefetchable function call statements and the fault address is in between the two call statements. Then only the second call statement needs prefetching. If we do not have addresses of the two call statements, it is not easy to prefetch the second function call only rather than both of them in the prefetch block. To prevent AP from misprefetching function calls, the position of a callee function is used when the fault manager searches for a proper position to enqueue accurate information in the prefetch queue. This type of label contains valuable position information for it..

Searching a proper block is simple as the nodes in the list are ordered in ascending of the starting address of each block. Each blocks' start address is compared to the faulted address and if it does not fit the search goes on to the next block. When a page fault is incurred, the prefetching table is searched by the page fault manager to enqueue some prefetchable pages. There are two different schemes of prefetching depending on the prefetch block. The first is for primary prefetch blocks which may need just a one block lookahead prefetching if it is on a page border. The second is for prefetch blocks which require prefetching for different pages according to the array info[ ] in the prefetch block. The major function related to this is called *find_func_pc()*. The function looks for a faulted function in the prefetch table and if there are some prefetch blocks which are lower address than the faulted address it skips them. Now, if these are no more with lower addresses, this is the right position to prefetch prefetch blocks. If it looked for a right position *find_func_pc* enqueues the addresses of objects in the info[ ] in the prefetch block which is shown in Figure 5.22. The number queued is limited by

PREFETCH_NUM which is set to 5 in the current version but it should be a function of memory size and the number of addresses in the info[ ]. The size of the queue is one of the critical factors influencing the performance of prefetching as well as the amount of memory pollution.

5.6.3 Running the Simulator

The hypothetical procedure of handling a page fault in AP on a real machine is shown in Figure 5.23. This figure is different from the simulation described in the previous sections in one way. The simulator does not have the page table, physical memory and secondary memory separately but has dummy memories having page table functionalities. However, this model is sufficient to explain the operation of AP in steps. The scenario in the figure is as follows:

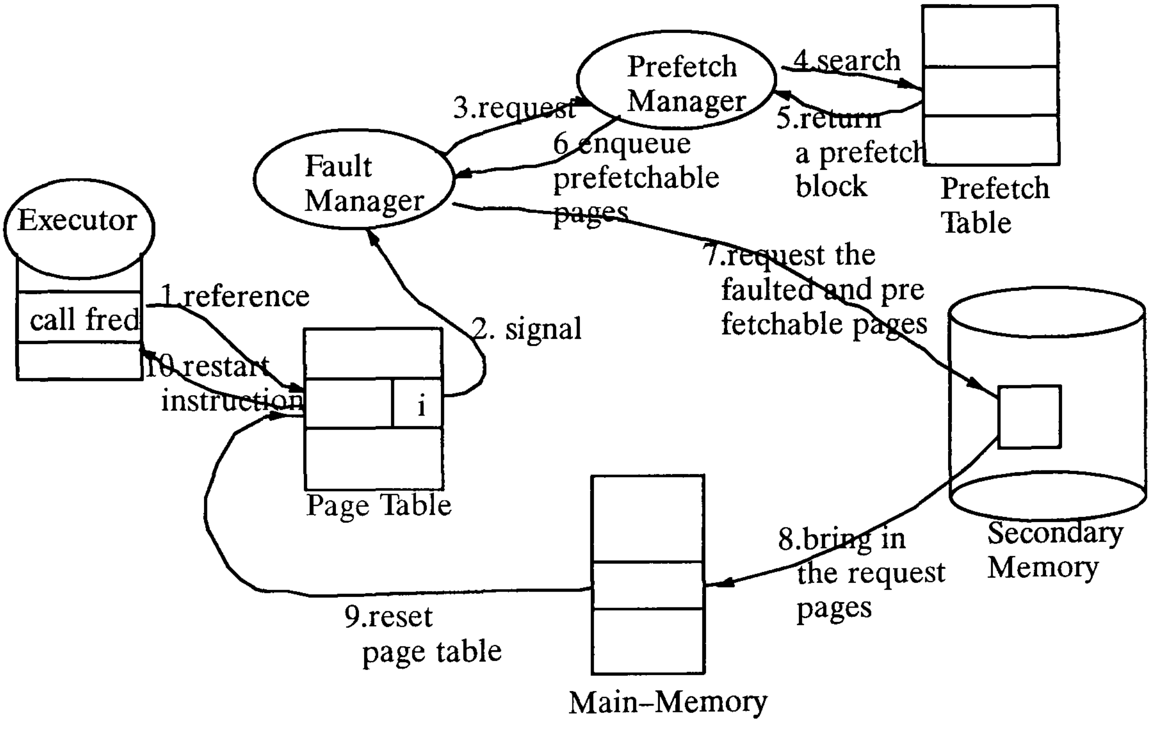


Fig. 5.23 The procedures in managing a page fault

1. If a program running on the executor tries to access a page having a function fred() which was not brought into memory, then a page fault signal will occur.

2. Then we check the page table to determine if the reference was valid or invalid. If it was invalid, the program is suspended.

3. The address evaluator raises a fault signal and this wakes up the fault manager.

4. The fault manager asks the prefetch manager if there are some prefetchable pages with the faulted address.

5. The prefetch manager looks up a prefetch block in the prefetch table for the program and adds the prefetchable pages to the queue and returns to the fault manager.

6. A request for the faulted page and prefetchable pages is transferred to the disk server and then the pages are scheduled for input.

7. When the disk read is completed, the pages are in main memory and page table is modified to show that the page is now in memory.

8. Restart the instruction that was interrupted by the fault address. (There is a method to restart the instruction in the faulted page before all the prefetched pages are read into the main memory)

The details of simulator related operations such as loading the prefetch table, searching for a proper prefetch block are omitted in the above description. However, an important operation for data prefetching is not described in the above procedure. The following section describes the details of the implementation of object data prefetching.

5.6.4 Object Data Prefetching Using Parameter Passing

Object data prefetching can be realized by prefetching encapsulated object data when one of its member function is invoked. The object data argument passing scheme is an important point when performing object data prefetching. The first part of this section discusses argument passing schemes in C+ + and then the implementation detail of the object data argument is described.

As stated in Section 5.2.5, an activation record contains storage for a function invocation on the program's runtime stack. *Formal arguments* are described as the list of arguments in a function definition and they are stored within the activation record.*Actual arguments* of a function call are the expressions used in the function call one to one mapping. Argument passing is the process of initializing the storage of the formal arguments by the actual arguments[Lippman 89]. Parameter passing semantics are an important issue in designing object oriented systems. For instance, Smalltalk adopted a natural parameter passing method, *call by sharing*, through passing a reference to the argument object. The same mechanism was used in CLU and in Emerald, a distributed object oriented system, with a different name where it is called *call-by-object-reference*. However, the default initialization scheme of parameter passing in C++ is *pass-by-value*, take an copy the rvalues (data value which stored at a location in memory), of the actual arguments into the storage of the formal arguments. Two alternatives to pass-by-value are *pass-by-pointer* and *pass-by-reference*. In these cases, the formal arguments are declared as pointers or type references. A reference argument passes the lvalue, a location value where its data value is stored, so as to modify the actual argument and not a local copy.

Although prefetching the argument object is impossible to realize at compile time as shown in Brent's work [Brent 87], it can be implemented at runtime because we make use of runtime symbols generated by a debugger. To prefetch the arguments passed to a function at runtime, AP invokes some extended functions of GNU debugger to get the location of argument objects which are passed by pointers or references. As stated earlier, the relationship between a function call statement and its arguments cannot be built at compile time because the location of the arguments are unspecified and they used to be referenced by pointers. However, the requirement to measure performance caused by accurate prefetching of argument objects is quite substantial so as to find the effect of the separate storing of encapsulated object's data and member functions in different pages. Therefore, this scheme is quite expensive to implement at runtime but it is worth investigating to measure it regardless of the cost.

While the simulator is running, if the executor encounters a call instruction it sets a flag to take a record of addresses of object's data which are passed as actual arguments to the function. Then the executor lets the CPU process the instruction one step and then the executor takes a record of all the position information about the arguments passed for prefetching them. For the pass–by–value scheme, the location information which is used to take a local copy of the argument objects are saved and they are added to the prefetch queue. When the function is initiated, the addresses are used for performing the copying process. So, before the copying process happens, the page containing the object can be prefetched if it is not resident in s_main memory. For the pass–by–pointer and pass–by–reference, the address of the pointers involved in the argument passing operation can be extracted out by the extended debugger functions. This is an indirect operation which reads addresses of the arguments using the contents of the pointer and gets the addresses of the objects. If the addresses are taken from the pointers, then they are added to the prefetch queue. The rest of the prefetching operations for pass–by–pointer and pass–by–reference are the same as for in pass–by–value, thus, the position information in the prefetch queue will be prefetched like the pages from the prefetch table.

5.7 Summary and Discussions

To summarize the description of the AP implementation, the virtual memory simulator comprises of the executor, s_main memory, s_secondary memory, prefetch queue manager and page fault manger. The virtual memory simulator takes prefetching information from the *pbfinalout* file which is generated by the compiler. The sequence of generating the prefetching tree which will be used by prefetching manager in the virtual memory simulator is as follows.

1 build a program skeleton by control flow analysis

2 build class inheritance trees

3 collect all object data and variables which are related to each function.

4 link a class name and its member function names

5 combine all the trees into a prefetch tree.

6 get all relocated addresses of all objects

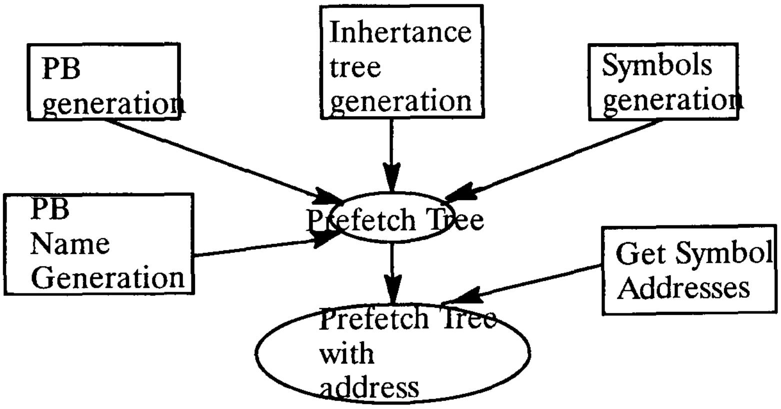


Fig. 5.24 Process of building a prefetch tree.

All the procedures except 6 take place concurrently at compile time whereas 6 is performed independently at link time. Using the prefetch tree, the virtual memory simulator searches an associated prefetch block with a page fault and prefetches them if they are not found in s_main memory. Figure 5.24 shows a diagram of the the prefetch tree generated by the compiler. PB generation corresponds to 1 and PB name generation means the operation described in Section 5.24. Figure 5.25 illustrates the final

```
main 2f36
p1 p2 p3 __$tmp_0

L20107 2f50

FC point_PSpoint_SI_SI 2118
 NSTD _malloc 4904
 NSTD __builtin_new 4468

L20108 2f62

FC point_PSpoint_SI_SI 2118
 NSTD _malloc 4904
 NSTD __builtin_new 4468

L20109 2f6c

FC rectangle_PSrectangle_Spoint_Spoint 29e6
 NSTD _malloc 4904
 NSTD shape_PSshape 26e0
 NSTD append_PSspgslist_PSshape 2550
 NSTD append_PSslist_PV 22aa
```

Fig. 5.25 The final prefetch tree generated by the compiler

```
name=main, st_addr=2f36, ed_addr=3012
  info = 2f36
name=L20107, st_addr=2f50, ed_addr=3006
  info=2f50   info=2118   info=4904   info=4468
name=L20108, st_addr=2f62, ed_addr=3006
  info=2f62   info=2118   info=4904   info=4468
name=L20109, st_addr=2f6c, ed_addr=3006
  info=2f6c   info=29e6   info=4904   info=4468
  info=26e0   info=2550    info=22aa
```

Fig. 5.26 The final prefetch list in the simulator

prefetch tree in a *pbfinalout* which is the final product of the compiler. When this *pbfinalout* is read into the simulator for running, the prefetch table becomes a linked list having entities shown in Figure 5.26. For example, the start and end addresses of the main function are illustrated in the figure and this first prefetch block does not have any prefetch information. The second prefetch block starting with name L20107 ranges from the position of this function 2f50 to 3006 and it has three prefetchable pages. The third and fourth names have different starting address but the same end address. This means that these names belong to a prefetch block but they represent positions of function calls in the block.

A problem posed during the implementation is that if library functions are invoked their locations are specified but sub functions called by the library functions are not processed in this version because library source code is not available. Thus, when the page fault manager looks up the prefetch table for builtin library functions to collect the function addresses, they can be found easily. However, in this case, prefetching information for nested functions in the builtin function cannot be found. The prefetching compiler was designed to be able to process even library functions but recompilation of the whole C+ +library needs to be carried out in the future.

The other problem is that the timing of prefetching pages is critical. When some pages are read into main memory and they are not referenced soon the effect of memory contamination is as severe as in most other prefetching systems. When a group of pages is prefetched too early to be referenced, because the number of prefetches (PREFCH_NUM) is too big compared to the s_main memory size, some of them may be not referenced while they stay in s_main memory. This kind of over prefetching breaks the working set. For example, if a real reference follows after the page was purged out, then the misprefetched page breaks the working set and must be read in again. Consequently, the necessary formula for this is strongly related to time. Thus, AP does not mean a prefetching of a set of pages which will be referenced in the long term but only pages which will be referenced in the near future. This is a function of time t, the pages of main memory reference string $R(t)$ and the number of pages in main memory at a given time $z(t)$. As a further study, the effect of mis-prefetched pages in a virtual memory system has to be clarified in algebra.

Chapter 6
Performance Measurement and Analysis

The last three chapters have described how prefetch trees may be constructed using the properties of an object-oriented programming and control flow analysis. This chapter discusses several performance measurement schemes for virtual memory systems. Then we describe some performance measurement and analysis using example programs in order to show how the AP mechanisms described in earlier chapters effect the overall performance of virtual memory systems.

This chapter begins by evaluating the performance of the compiler which generates a prefetch table in terms of time taken to compile. A comparison is made between the original GNU C+ + compiler and the extended one. The following section discusses two general performance measurement methods for prefetching based virtual memory systems. Total system performance of AP is described in the following section, as well as the limits of AP in terms of fault management. The next section discusses what kinds of faults can be managed and those that cannot. The final section discusses various points concerning the performance of AP and its influence on the whole computer system.

6.1 Performance Measurement of the C+ + Compiler

This section discusses simple tests which illustrate the performance of the experimental implementation of the system – the modified GNU C+ + compiler. The compiler performance has not been optimized yet but it is worth comparing with the original GNU C+ + compiler (version 1.32.0) in terms of the time taken to compile and link so that we can measure the time spent in generating a prefetch tree.

The tests for the extended C+ + compiler which generates a prefetch tree for AP were performed on a SUN® 3/60 Workstation that has four megabytes of memory and runs the SUN implementation of the Berkeley BSD 4.3 UNIX operating system (version 4.0.3).

The extended C+ + compiler can compile any C+ + program and it generates exactly the same executable code as the GNU C+ + compiler does. Two small C+ + programs are used throughout the performance evaluation. The first example program is *myshape* from Stroustrup's book. This program is relatively small but uses many object–oriented features. The source code of the program is 360 lines of C+ + code. The second test program is Marshall's *LRing* queue manipulation program which has 605 lines of C+ + code.

To illustrate the performance of this compiler, parse time and link time were shown in Table 6.1 and Table 6.2. Most of the extensions (such as building a program skeleton, inheritance trees, collecting object data) for prefetch tree generation are included in the parse pass. Table 6.1 shows that the overhead of parse time in the extended compiler is 1.16 times than the original. This amount of overhead is quite acceptable.

	v.1.32.0 ORG	v.1.32.0 AP	v.1.32.0AP v.1.32.ORG
LRing	9.20 sec.	10.88 sec.	1.18
Myshape	9.06 sec.	10.46 sec.	1.15

Table 6.1 The parse time of extended GNU C+ + compiler

	v.1.32.0 ORG	v.1.32.0 AP	v.1.32.0AP v.1.32.ORG
LRing	1.82 sec.	12.94 sec.	7.10
Myshape	1.74 sec.	13.12 sec.	7.54

Table 6.2 The link time of extended GNU C+ + compiler

As stated in the previous chapter, the linker was also extended to link prefetch tables in several files including library files into a final prefetch table and then get addresses for the symbols in prefetch blocks such as objects and variables. Linking time was measured (Table 6.2) so as to observe the time spent compared with that of the original linker. As we can see in the table, the extended linker takes about 7 times longer than the original. This is because linear searching for addresses for every symbol takes time proportional to N^2 / 2 (where N is number of symbols). This overhead can be reduced by an optimization such as using hash table for symbol lookup but it is still likely to remain as a relatively big overhead of the compiler.

6.2 Performance Evaluation of AP

This section describes how performance of AP can be evaluated. Performance evaluation for a prefetching system is different from that of demand fetching or OBL because it brings in dispersed pages. This section begins by discussing how to evaluate a prefetching policy and what points ought to be considered in the evaluation. The following section discusses lifetime curves and a space-time product developed for prefetching policies. The section after discusses the performance evaluation policy adopted in this simulation, and is followed by two performance tests for the C+ + programs mentioned in the previous section.

6.2.1 Cost Measurement Method for Prefetching Policies

It is important to consider how the performance of prepaging policies should be evaluated. In particular, how general prefetching paging systems are effected by fault rate is important. Horspool[Horspool 87] pointed out that the following points of virtual memory system influence operating system performance:

i) Amount of main memory that is occupied by the program.

ii) Number of page fault interruptions.

iii) Number of pages loaded into main memory.

iv) Number of pages removed from main memory.

v) Total page wait time.

iii) and iv) should be equal to the number of faults in conventional fixed or variable space demand paging systems. The total page wait time should be approximately proportional to the number of faults. Therefore, we can reasonably describe the performances of fixed space and variable space policies by fault rate curves that show the trade off between the main memory allocation (mean memory allocation for a variable space policy) and the numbers of faults. This can be determined from the lifetime curve which gives the mean number of references between faults when the mean resident set size is given. A knee in the lifetime curve which is shown in Figure 6.1 is a maximum point of mean lifetime and a minimum point of real space time product for a given size of resident set [Denning 79].

Also, the *effective access time*[Silberschatz 88] for a demand paged memory has significant influence on the performance of a computer system. Let p be the probability of a page fault ($0 < p << 1$), and memory access time C_f. We would expect p to be very close to zero. The effective access time is then:

$$\textit{effective access time} = (1 - p) \bullet C_f + p \bullet \textit{page fault time.}$$

As long as we have no page faults, the effective access time is equal to the memory access

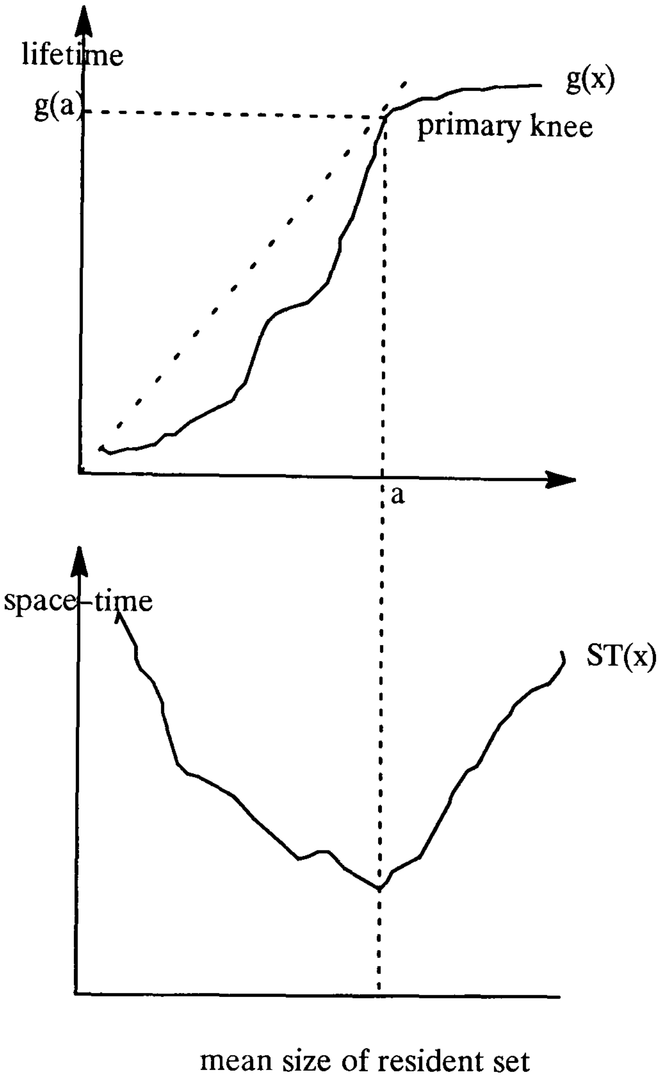


Fig. 6.1 Life time and space time curves

time. Here, we can see that the effective access time is directly proportional to the page fault rate.

When prepaging is considered, the total number of page fetches is no longer directly proportional to the number of page faults. Because the number of prefetched pages is determined by the prefetching policy, in part, by the prefetch tree information and, in part, by the length of prefetching queue in AP. Also, it is unreasonable if prefetching a page are directly comparable with the costs of demand fetching a page. Other researchers have sometimes assumed that each prefetch incurs a cost equal to about 20% of the cost of a demand fetch in OBL with a moving head disk as a secondary memory[Horspool 87].

A space–time product could represent the performance of a prefetching scheme the most accurately because it considers the five points listed earlier in this section. The relationship between fault rate and real space time product (RSTP) was formulated by Horspool[Horspool 87] and the following is taken directly from his paper.

$$\text{RSTP} = \int \text{SIZE(t) dt},$$

where SIZE(t) is the amount of real memory used by a program.

If the total real execution time of the program is T_r, then,

$$T_r = T_v + T_w,$$

where T_v = program duration in virtual (or process) time and T_w = total page wait time (time that the program is inactive waiting for page fetches to be completed). Thus,

$$\text{RSTP} = M_v \bullet T_v + M_w \bullet T_w,$$

where M_v and M_w represent the mean memory allocations over virtual time and during page wait time. Assumes that $M_v \approx M_w$,

$$T_w = F \bullet C_f + P \bullet C_p,$$

where F = number of page faults, C_f = expected time required to fetch a missing page, P = number of prefetches performed and C_p = expected extra time required to prefetch a page. Therefore,

$$\text{RSTP} = M_v \bullet (T_v + F \bullet C_f + P \bullet C_p)$$

When we wish to compare two paging policies, P_1 and P_2 on the same hardware they have identical values for M_v, So,

$$\text{RSTP}_1 - \text{RSTP}_2 = M_v \bullet [(F_1 - F_2) \bullet C_f + (P_1 - P_2) \bullet C_p],$$

(where RSTP_1, F_1 and P_1 represent the observed values of RSTP, F and P when using policy P_1 similarly for policy P_2) Simple rearrangement leads to the following result:

$$\text{RSTP}_1 > \text{RSTP}_2 \quad \text{iff} \quad (F_1 - F_2) \,/\, (P_2 - P_1) > C_p \,/\, C_f \quad (P_2 > P_1)$$

The ratio $(F_1 - F_2) / (P_2 - P_1)$ shows a ratio of successful prefetches or an effective faults decreasing. It represents the relative cost of prefetching a page versus the cost of demand fetching a page and usually it is less than 0.2 in OBL but it would slightly higher than this in AP.

To compare demand paging and AP, Hospool's model needs a slight modification. In practice, Cf consists of disk seek time and operating system overhead. Smith[Smith 78] reported that operating system overhead would be a dominant factor in any system where the CPU is fully occupied. In particular, it is obvious in process migration system. So, in demand fetching, $Cf = Dsf + Of$ (where, Dsf = disk seek time and latency delay for a normal fault, Of = operating system overhead such as context switching, I/O initiation). However, in the case of demand prefetching, operating system overhead is less dominant because several pages are fetched at the same time. Thus, the operating system overhead in demand prefetching does not directly proportional to the number of page fetching. Cp for demand prefetching, like AP, needs less accounting for operating system overhead. So, $Cp = Dsp + Op$(where Dsp is disk seek time and latency delay for prefetching page, Op = operating system overhead for prefetching a group of pages). Therefore, the *effective decrease* for AP is $(Dsp + Op) / (Dsf + Of)$. Moreover, when AP prefetches several pages without memory pollution, the benefit of AP should be more than that and it is a near optimal policy. If we consider page sorts in the disk I/O routine in the UNIX operating system, the assessment of a prefetching policy becomes more complicated.

Consequently, as we can see in the above performance evaluation techniques such as effective access time, Hospool's cost measurement method and Denning's life time curve, fault rate is the most significant factor and memory access time is directly proportional to the fault rate. Therefore, the performance tests in the following section are mainly concentrate on measuring the fault rate.

6.2.2 Policies on Performance Measurement for AP

The policy adopted for evaluating the performance of AP in the following section is quite restricted compare to Hospool's theoretical method which was described in the previous section. This is because of limitations in the functionality of the simulator with respect to measuring some time parameters. The simulator is unable to measure times such as: the total execution time of a process, the expected time required to fetch a missing page, the expected extra time required to prefetch a page, disk seek time and latency delay, and the operating system overhead. These time parameters can be obtained when AP is implemented on a real machine.

However, most of the times involved in virtual memory management system are closely associated with page fault rate. For instance, the total page wait time and the overhead from the operating system should be approximately proportional to the number of faults. This hypothesis is obvious in randomly accessible memories like a cache or a RAM disk. We can therefore reasonably assess the performance of AP in terms of paging rate.

Another point that we have to consider in the performance evaluation of a virtual memory system is whether it implements a local or a global memory management method. A global memory management policy seeks to optimize the memory of the entire system rather than on a per process basis. However, this can lead to a paging system so complicated that it can not be modeled at all [Breecher 89]. The performance evaluation in the following is therefore limited to a per process memory management scheme.

6.2.3 Performance Measurement of AP

To illustrate the performance of the AP, prefetch tables and executable files which are generated by the extended C+ + compiler for the test programs are used in the simulator. A number of tests were made which involved recording the number of page faults and page fetches. However, the time taken to execute the test programs cannot be measured because the simulator was not designed to measure such times. As stated earlier, the simulator runs on top of UNIX and, therefore, all disk operations are hidden under the operating system and it is quite difficult to measure the execution time accurately without putting all the functionality of AP into the UNIX kernel.

Results were measured for several main memory sizes which are usually within virtual memory operating ranges. S_main memory size varied from approximately 20% to 80% of the whole addressing space of a process. This is the working range for conventional virtual memory systems. Each page size was set to 512 bytes in this simulation. Prefetching is also dependant on the size of the prefetching queue, so this is another variable in the simulation.

Table 6.3 and 6.4 show the basic performance characteristics in terms of the amount of paging. The first column shows the sizes of s_main memory and the numbers in bracket are percentages of the s_main memory size out of the total pages required to run the program. The second column is the size of the prefetch queue and it is only applied to the AP scheme. In the third and fourth columns, paging for On-Demand (OD) fetching and OBL in terms of the number of page faults and total pages read into main memory were measured to compare them with the result of AP. Also, the number of pages read in by the OD policy is the same as its page faults and it provides a reference for comparing the paging rates of the other two policies. This is the minimal paging rate that we can expect in a paging system. Non prefetching systems are unable to suppress page faults less than OD's number. The results for OBL are shown at the fourth column in order to compare its results to OD and AP. The first sub-column in OBL is the number of page faults and the second sub-column is the number of the total readin pages into main memory. In the fifth column, the details of paging by AP are shown in terms of the number of faults, the amount of prefetching, the total amount of paging (the sum of the number of faulted pages and prefetched pages) and the prefetching percentage which is the number of prefetched pages out of the total pages read into main memory. The sixth and seventh columns illustrate page fault ratios comparing AP to OD fetching and AP to OBL. Finally, the last column shows Hospool's RSTP in Section 6.2.1 of AP compare to OD.

The test results for the *LRing* program are given below in Table 6.3, Figure 6.2 and Figure 6.3. The size of the executable file of the test program is 160k bytes but its real referenced address space during execution is 22k (44 pages), the rest being functions which are not invoked during execution, symbol table and dummy pages which are built by the UNIX executable file format. As far as paging rate (number of pages read into s_main) is concerned, the overall result of OBL is almost double of OD if it is estimated just by page movements. However, when the access time is considered as an another factor, the performance of OBL may be improving.

memory size k bytes, pages, (%)	#pref que.	OD	OBL	AP				AP OD	AP OBL	
		fault	fault/pgin	fault	pref.	page in	pre-fetch %	fault ratio	fault ratio	RS–TP
5k, 10, (23)	3	356	438, 835	333	51	384	**13**	0.93	0.76	0.62
5k, 10, (23)	6	356	438, 835	331	68	399	**17**	0.92	0.76	0.37
7.5k, 15, (34)	6	216	248, 454	193	29	222	**13**	0.89	0.78	0.79
10k, 20, (45)	6	154	164, 288	139	15	154	**9.7**	0.90	0.79	1
12.5k, 25, (57)	6	92	111, 189	82	12	94	**12.8**	0.89	0.74	0.83
15k, 30, (68)	6	76	74, 126	67	12	79	**15.2**	0.88	0.71	0.75
17.5k, 35, (79)	6	43	62, 104	34	10	44	**23**	0.79	0.51	0.9
25k, 50, (114)	6	43	34, 59	34	10	44	**23**	0.79	100	0.9

OD: On Demand Fetching,
OBL: One Block Lookahead,
AP: The Accurate Prefetching,
prefetch (%) = #prefetching / #pagein in AP * 100,
fault fatio = AP fault / OD fault

Table 6.3 Page fault and prefetching ratio for *LRing*

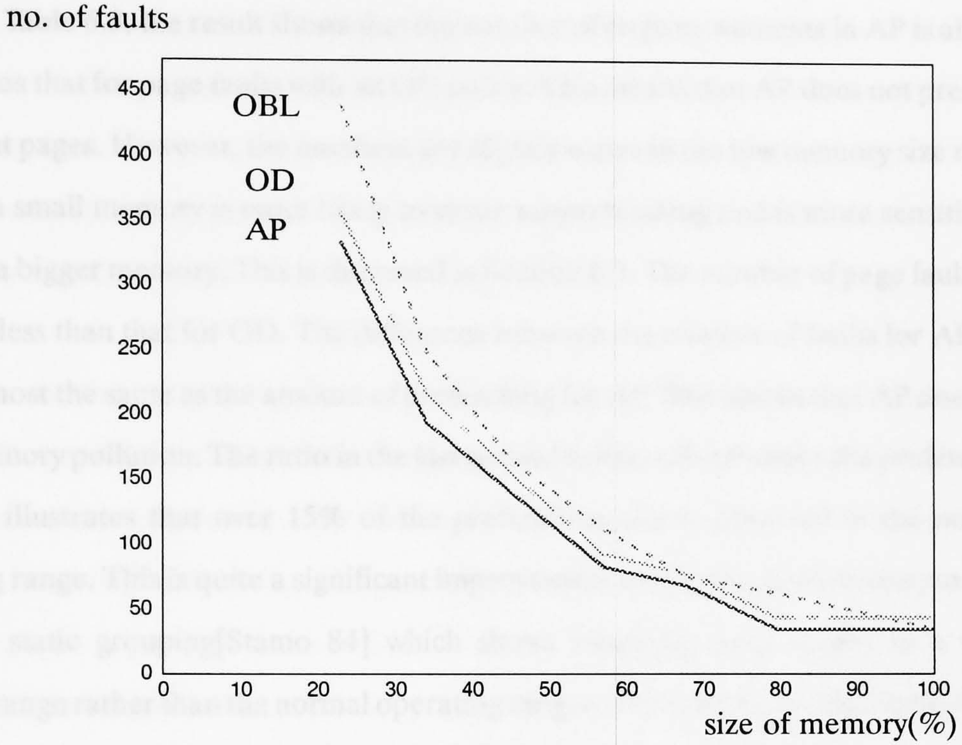


Fig. 6.2 Page fault reduction for LRing

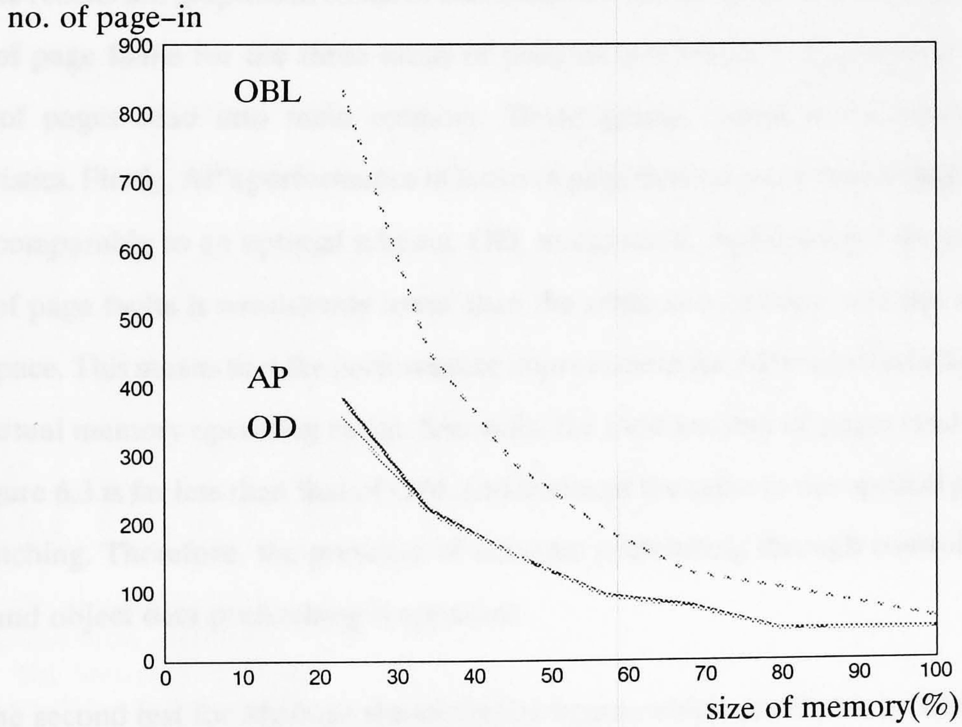


Fig. 6.3 Number of page fetches for LRing

In Table 6.3, the result shows that the number of page movements in AP is almost the same as that for page faults with an OD policy. This means that AP does not prefetch redundant pages. However, the numbers are slightly worse in the low memory size range because a small memory is more likely to cause misprefetching and is more sensitive to this than a bigger memory. This is discussed in Section 6.3. The number of page faults for AP is far less than that for OD. The difference between the number of faults for AP and OD is almost the same as the amount of prefetching for AP. This means that AP does not cause memory pollution. The ratio in the last second column of AP shows the prefetching ratios. It illustrates that over 15% of the prefetching rate is obtained in the normal operating range. This is quite a significant improvement for paging systems compared to Stamos's static grouping[Stamo 84] which shows relatively good results in a small memory range rather than the normal operating ranges. The last column illustrates RSTP which is worth comparing to Smith's 0.2 in OBL discussed in Section 6.2.1.

The results are graphed in terms of two different values: Figure 6.2 illustrates the number of page faults for the three kinds of policies and Figure 6.4 shows the total number of pages read into main memory. These graphs reveal some interesting characteristics. Firstly, AP's performance in terms of page faults is much better than OBL and it is comparable to an optimal scheme, OD, as expected. As Figure 6.2 shows, the number of page faults is consistently lower than the other two policies over the whole address space. This means that the performance improvement for AP is significant for the overall virtual memory operating range. Secondly, the total number of pages read in by AP in Figure 6.3 is far less than that of OBL and is almost the same as the optimal points of OD fetching. Therefore, the presence of accurate prefetching through control flow analysis and object data prefetching is apparent.

The second test for *Myshape* shows similar figures which are shown in Table6.4, Figure 6.4 and Figure 6.5. The size of the executable file of the benchmark program is

135k bytes but its image after stripping the symbol table is 57k bytes. The symbol table is only used for indexing objects in the simulation but not all of them could be used if AP is implemented on a real machine. The real memory reference address range is smaller than this, only 17k (34 pages) because, again, some pages are dummy. The rest of the test conditions are the same as the previous measurement.

The results obtained again support the performance improvement through AP. Thus, fault rate and page transfers are far ahead of OBL in accuracy. One notable point in this test is that the AP prefetching ratios for the same memory size, i.e. both 29.4 %, are different from each other depending on the prefetch queue size. For example, the prefetch rate for a prefetch queue of 6 is almost double that of queue of 3. However, the accuracy is relatively low if the prefetch queue size is unnecessarily big and the proportion of prefetched pages at a fault is too high compared to the number of existing pages in main memory.

memory size k bytes, pages, (%)	#pref que.	OD	OBL	AP				AP OBL	AP OBL	RS–TP
		fault	fault/pgin	fault	pref.	page in	pre-fetch %	fault ratio	fault ratio	
5k, 10, (29.4)	3	254	280/491	231	31	262	**11.8**	0.91	0.83	0.74
5k, 10, (29.4)	6	254	280/491	215	67	282	**23.7**	0.85	0.76	0.58
7.5k, 15, (44)	6	96	152/254	81	15	96	**15.6**	0.84	0.53	1
10k, 20, (59)	6	57	66/115	47	11	58	**19**	0.82	0.71	0.9
12.5k, 25, (74)	6	38	45/78	29	9	38	**23**	0.76	0.64	1
15k, 30, (88.2)	6	34	28/49	25	9	34	**26**	0.74	0.89	1

OD: On Demand Fetching,
OBL: One Block Lookahead,
AP: The Accurate Prefetching,
prefetch (%) = #prefetching / #pagein in AP * 100,
fault fatio = AP fault / OD fault

Table 6.4 Page fault and prefetching ratio for *Myshape*

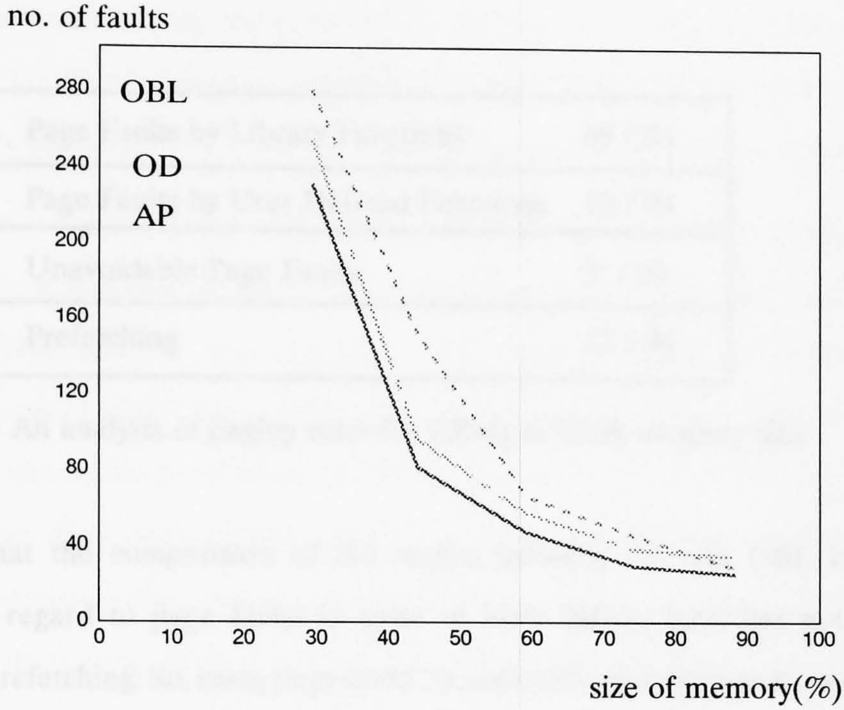


Fig. 6.4 Page fault reduction for Myshape

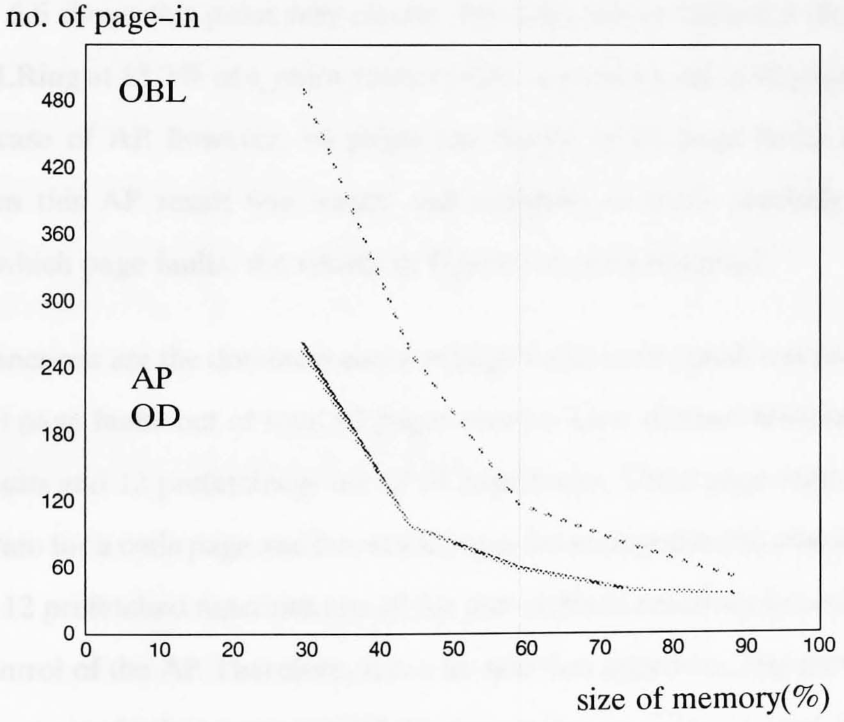


Fig. 6.5 Number of page fetchings for Myshape

Page Faults by Library Functions	69 / 94
Page Faults by User Defined Functions	10 / 94
Unavoidable Page Faults	3 / 94
Prefetching	12 / 94

Figure 6.6 An analysis of paging rates for LRing at 12.5k memory size

Notice that the comparision of the results between AP and OBL is quite significant with regard to page faults in spite of basic library functions not being recompiled for prefetching. So, many page faults caused by library functions are currently unavoidable. The current version of AP is only able to generate a prefetch tree for functions programmed by users and that is why AP's performance limit still remains. For example, Figure 6.6 shows this point very clearly. The fifth row in Table 6.3 illustrates paging rates for LRing at 53.2% of s_main memory size. It gives a total of 92 page faults by OD. In the case of AP, however, 94 pages are readin by 82 page faults and 12 prefetches. When this AP result was traced and analyzed to know precisely which functions made which page faults, the results in Figure 6.6 were obtained.

Library functions are the dominant cause of page faults in the small test program. They incurred 69 page faults out of total 94 pages read in. User defined functions also cause 10 page faults and 12 prefetchings out of 94 page faults. Three page faults at the start of the program for a code page and two stack pages are included in the unavoidable page faults. The 12 prefetched functions are all for user defined functions because they are under the control of the AP. Therefore, it can be said that approximately more than 50%, (12 user page prefetches among total 22 user pages read in) of user defined functions are accurately prefetched by AP. Consequently, if all C+ + library functions (including C library functions) were recompiled and we were able to generate the prefetch tree for the library functions, more than half the total page faults would vanish because the principle of the AP is the same for user programs and library functions.

The results of object data prefetching described in Section 3.5.2.2. are shown in Table 6.5. When we measure the effect of object data only, the test results show that average prefetching ratio is approximately 3.1%. The influence of object page prefetching in a small memory is more than that in a large memory size. This is because the potential for object data prefetching by AP is higher in the former. Thus, object data prefetching occurs when its member function incurs faults and the object data is not stored in main memory. If the member function is in main memory then its object data is not prefetched but is demand fetched by a fault. So, the possibility of both member function and its object data not being in main memory is higher for a small memory system than for a larger memory system. Another reason for the low rate prefetching rate is that the candidate of prefetching is quite limited since object data prefetching is only for member functions which have object data as transferred argument. Some member functions like constructors are not included in the candidate of the object data prefetching function because the objects are not created yet. Also, most of library functions are also excluded from the candidate because only user defined member functions are considered for the candidate of the object data prefetching in this simulation.

6.3 Manageable and Unmanageable Faults by the AP

Many different kinds of factors cause page faults in virtual memory in local or distributed systems. Initial loading of a program, dynamically created objects in heap or stack, sequential programs in different pages and conditional or unconditional control branch by jumps or function calls are the major reasons for page faults. Among these, some can be made manageable and be controlled to inhibit faults by the strategy in AP

memory size kbytes, pages, %	Object Data Prefetching fault	Prefetch (%)
5k, 10, (23%)	14/340	4.1
7.5k, 15, (32)	6 / 210	2.9
10k, 20, (42.5)	6 / 146	4.1
12.5k, 25, (53.2)	2 / 91	2.2
15k, 30, (64)	2 / 75	2.7
17.5k, 35, (74)	1 / 42	2.4

Table 6.5 The effect of object data prefetching for *LRing*

for optimal space–time product. For instance, AP can be used efficiently for initial loading of a program and at every phase transition in the working set so as to supress the number of consecutive faults which used to be managed by loading a segment in a segment–paged scheme. Moreover, inherited constructors or destructors are under the control of AP and some object data can be managed by run–time AP. Although it is not implemented in this simulation, static grouping of encapsulated objects and virtual functions in the same class hierarchy may also reduce pagings.

On the other hand, some faults are controllable but at too great on expense. For example, dynamically created object data may be prefetched when its member functions are read into main memory but much location information for objects would need to be kept at run time. Also, some page faults can be managed if the size of the prefetching queue is increased. The parameter which decides the number of pages prefetched at the same time influences AP performance enormously. The bigger the parameter, the more pages will be prefetched. Because the number of addresses queued in the prefetch queue is increased. However, this does not make for a good prefetching system because it could commit memory pollution or purge some useful pages from main memory to secondary memory. An example of this is when a function is dispersed in several pages that they cannot all be prefetched at the same time because the priority of the page which contains a return statement of the function is lower than the page having the function head. If any low priority pages are prefetched into main memory, some higher priority pages will be purged out. The priority can be adapted for every page to be read in but this is not considered by this implementation. If these pages cause a page fault, it should considered uncontrollable.

6.4 Discussion

In this discussion, several ways to improve AP are considered. The first is the question of what is a proper memory size for a prefetching system and how to use it efficiently. The second is how to assign priority to prefetched pages compared to demand fetched pages. The third is the effect of branch prediction to eliminate the obstacle of conditional branches. Finally, the influence of AP on disk sorting is considered.

With regard to the first, there is a prerequisite on an ordinary non-accurate prefetching scheme that for effective prepaging you have many memory pages. Thus, if devoting a page to prefetching sacrifices more than about 1% of your memory, then you lose – the increased misses caused by the smaller memory will not be compensated by successful prefetches. On the other hand, if you have more than about 100 pages, devoting one to holding prefetches is worthwhile – it doesn't cost you many misses, and it will prefetch a useful page often enough. So, it seems that if a system has about 100 units of memory (pages in a main memory, or cache blocks in a cache), then prefetching is worthwhile it. Consequently, if a system has significantly fewer pages/blocks, it is better off devoting them all to normal caching by demand fetching. However, if a system has significantly more, it is better to spend a few on prefetching. AP does not need to obey the prerequisite and even reverses it because it depends on accurate prefetching and works well even in small main memory systems. Some results in the previous section revealed slight performance degradation in the small memory because overprefetching was intentionally generated (60% -six prefetch pages out of ten pages in main memory). However, AP works very well even if 40% of pages are devoted for prefetching and it proved that even such a large page allocation for prefetching is still useful.

In answer to the second question, it has been found beneficial to give prefetched pages less chance to remain in a main memory than demand fetched pages. A similar philosophy is reflected in the prepaging policy of the VME/B operating system. However, these are not considered in this simulation because if AP prefetches pages accurately they are likely to be referenced in the near future. In this case, the usefulness of prefetched pages are the same as demand fetched pages. So, there are no reason to give less priority for prefetched pages in the case of a real accurate prefetching scheme.

One of major problems in designing AP is to ensure a steady flow of instructions. A change in the expected sequence of instructions due to a branch will cause us to reload a page. Since AP relies heavily on a prefetch tree which is generated by control flow analysis based on branch points at compile time, the size of a prefetch block is relatively small. A possible alternative to the approach we have taken is to predict control flows at branch points. If there is an accurate branch prediction algorithm available, the performance of AP is likely to be maximized when the prediction algorithm is combined with the prefetching scheme.

There are a number of branch prediction methodologies to reduce the performance degradation caused by branches. They are multiple instruction streams, prefetching branch targets, data fetch targets, prepare to branch, delayed branch, a taken not taken switch, and branch target buffer (The details can be found in [Lee 84]). Some of these replicate several branch targets and others try to prefetch a branch target by a special mechanism for calculating the target. The details of these algorithms are described in [Lee 84]. According to the results of the algorithms, the probability of correct branch prediction is about 70%.

As stated earlier, if a page is misprefetched into a main memory it could break the working set and, consequently, it may cause a number of page faults to reload the pages to form the working set again. The result of this kind of misoperation is so critical, therefore, it is unlikely to improve performance if any of the branch prediction algorithm discussed above are adopted into AP since their accuracy is about 70%, therefore, the other 30% could cause misprefetching. Moreover, branch prediction algorithms are mainly developed so as to improve time critical purposes such as pipelinings. However, AP does not rely so heavily on every branches compare to pipeline operations and is not so find like pipelining. It is a tradeoff between adopting a branch prediction algorithm where the probability is about 70% and a potential breaking of a working set by the rest of 30% of misprefetching. Therefore, no branch prediction algorithm is introduced to current version of AP but is worth investigation in the future.

Finally, disk sorting was briefly considered so that the accurate prefetching could reduce disk seek time. If an accurate prefetching system provided more than one object or page which may or may not consecutive each other to the disk scheduling queue, optimal scheduling of disk head movement could be achieved and consequently the seek time can be minimized. This strategy gives many benefits if it operates on randomly accessible secondary memory. It also shows the same or better performance than conventional fetching strategies in movable head disk based secondary memory system. This is because giving a number of prefetchable pages to a disk system having a long disk scheduling queue enables it to optimize disk access by ordering them [Seltzer 90]. Pages which are the prefetch queue can be used for efficient disk page sorting. In conventional disk based virtual memory system, disk page requests are sorted into ascending or decending order depending on the disk sort algorithm adopted in the system. For example, SSTF (shortest seek time first), SCAN and LOOK algorithms[Silberschatz 89] perform track sorting before real read/write operations are carried out. Therefore, if information on the prefetching pages is available, partial sorting can be more efficient because they provide lookahead information for future references.

Chapter 7
Conclusions and Directions for Future Work

It is a difficult task to realize an accurate prefetching system but this thesis has shown that it can be done if we make use of control flow analysis and the properties of object oriented languages. The notions and practices behind the use of control flow for *nearness* algorithms, such as restructuring or grouping of program pieces, were well known in the early stages of virtual memory development. However, the notion of prefetching objects for object oriented computing models is relatively new and is different from previous work. It allows objects to be prefetched accurately without any memory pollution and the performance of the memory management system is thereby significantly enhanced. This chapter concludes topics discussed in this thesis and indicates some of the possible areas for further research.

7.1 Conclusions

The requirements were to develop a scheme that could reduce fault rates as much possible so as to handle dispersed objects in object oriented languages in the latest memory systems which allow random block or page accessing (e.g. large RAM cache or RAM disks). This motivation defines the goal of this thesis to be the development of a system that can support an accurate prefetching methodology, lowering fault rates and solving the memory pollution problem for object oriented languages.

The limits of existing policies such as restructuring and grouping based on *nearness* algorithms were obvious when dealing with complicated and dispersed objects in object oriented programs. These policies do not have any mechanism for fetching objects at the time when they are required and do not fetch objects if they will not surely be referenced in the near future. Also, the restructured program often runs worse than the original non–restructured version since a different set of input data even for the same program can alter the trace dramatically. These approaches were inefficient in suppressing the set of consecutive faults that happens at every phase transition. Alternatively, OBL is a simple prefetching scheme for reducing disk access time but is hardly adequate to handle the latest memory systems which allow random accessings. Therefore, a new approach for tackling the problem should be able to control the fetching of objects by using information derived from source program structures. Moreover, as a means of lowering fault rates, a group of related objects can be tied and fetch them together. When one object in the group is demand fetched, the other objects will be prefetched.

To prefetch a set of objects or pages together, an intra and inter object relationship must be built to stick them to each other. We tried to find a means of supporting this facility among the properties of object oriented languages. Data abstraction and encapsulation enabled us to build intra relationships between operations and their data. These properties provided data dependencies to be able to tie them. This property was used to prefetch object data which were usually stored in different pages or segments to the functions. However, prefetching of data objects by encapsulation at compile time was difficult to implement because there were many constraints such as dynamic creation and the use of pointers. So, this was implemented at runtime to observe the effect of data object prefetching. Inheritance provided another good property for building an inter object relationship between objects in the same hierarchy. The *yoyo* problem is a typical example of high inter–object dependency causing busy control flow among objects in the same hierarchy because the execution of methods goes up and down the class hierarchy. Although there is a strong tendency to *yoyo* in user programs, only limited member functions such as constructors and destructors were worth prefetching because there were a few exceptions of the functions from the *yoyo*. Virtual member functions also showed the *yoyo* phenomenon but they were not worth prefetching all together because not all member functions in an object are used in the near future and we do not want to prefetch objects without some certainty that the object member function is referenced soon. Therefore, control dependences and data dependences by control flow analysis were required to reinforce the relationship.

Some inter and intra object relationships are expressed syntactically by object invocations. Control dependency of a program can be exposed by using the control flow analysis technique. Significant program structure was encoded in the compact form of a table, the program being divided into prefetch blocks which obey the single–entry–single–exit rule. In the control flow graph model, branch points became leaders for building prefetch blocks. This rule led to a guideline required for establishing prefetch blocks for high level statements containing branch points. A problem raised by this approach was that there were too many fine grained prefetch blocks which did not contain substantial prefetching information. These nonsignificant blocks are merged into their consecutive blocks. Data dependency implied by encapsulation also can be expressed using argument passing schemes. Object data were attached to the prefetch block containing the function call since they are very likely to be referenced in the near future by the function.

This extension of dependencies by control flow analysis revealed that the behavior of control flow analysis in early binding object oriented programming languages is almost the same as that in procedure oriented languages. Control transfer between member functions occurs through member function invocation and object data transfer occurs by parameter passing. This is because interfaces between objects are procedure basis. However, control flow analysis was not able to be applied to dynamic binding because the binding is not specified at compile time. Instead of this, dynamic binding can be used for static grouping of virtual functions. Therefore, the properties of object oriented programming provided some hints for the relationship which can be used for both static grouping of objects and prefetching related objects in a large memory system. However, these were not sufficient themselves to build the relationship for AP without the reinforcement of control flow analysis.

From the full implementation of the AP simulator, we can see that many page faults caused by function calls can be suppressed because the prefetch table provided reference information to the fault manager. This is a simple software approach to lowering fault rates. This result agrees with Portfield's findings quoted in [Gornish 90] which were that effective compile–time prefetching is often more effective than hardware prefetching because the compiler can analyse program structures. However, when the prefetch table was used by AP, prefetch operation was demand prefetching. So, a limit to suppressing of faults was remained, thus, whenever there was a page fault it could prefetch some object group but there was no way to prevent the fault itself by this demand prefetching scheme. This is a drawback of the software approach because no parallelism can occur between the prefetching operation and computation, therefore, and the fault rates cannot be zero.

The performance measurement for the compiler and the AP scheme were carried out using test programs and later the results were analyzed in detail. The simulator can measure paging rate but was unable to measure timing parameters. The measured results showed that about 20% of accurate prefetching was obtained in normal virtual memory working ranges. This means that the same amount of fault rates were reduced. However, this rate is expected to be easily increased into about 50% if all library functions related to a user program were recompiled by the AP compiler.

The results supported the original goal of this thesis in developing an accurate prefetching scheme for object oriented system without the drawbacks caused by those *nearness* algorithms or other prefetching schemes. In this, we have been successful in showing that AP achieved both a significant real improvement in performance, and was misprefetch free and relatively simple to implement. The key finding of these experiments is that the realizable AP substantially reduce the number of page faults caused by dispersed objects. AP provides not only significant improvement in reducing the number of page faults but also reduces memory pollution dramatically through very accurate prefetching. In particular, the page fault reduction for object data pages which have weak sequentiality is a valuable achievement. Some proportion of useless prefetches which could occur in OBL can be managed and page faults which were caused by long jumps can be suppressed by AP.

Accurate prefetching can be applied to many memory hierarchies although there are some minor problems if it is applied to cache memory. Since it needs to lookup the prefetch table, it may not be suitable for cache memory in conventional CPU, but if an associative memory is adopted to reduce the searching time by content searching of the table, AP can be used for cache memory as well. One significant advantage of AP when it is applied to local or distributed virtual memory system is that it reduces operating system and network overheads by moving a group of useful objects at the same time. Therefore, memory hierarchies such as cache, virtual memory, and distributed paging can be application areas of AP.

7.2 Future Work

We believe that the current AP implementation demonstrates the viability of this approach and meets our goal of enhancing virtual memory performance in terms of page or object fetching. However, a number of areas of future work have become apparent throughout the description of the mechanisms of AP. The first area is that AP should be implemented on a pure object-oriented system to evaluate performance variations. Although most of the main areas for individual object prefetching are considered in this thesis the implementation would meet more realistic problems which would have to be resolved. Optimization of the generation of the prefetch table by the compiler needs to be under taken and a real implementation of the accurate prefetching algorithm in an operating system would also need careful study.

Another extension of the accurate prefetching scheme is to adapt a static grouping for objects using the intra and inter object relationship. The use of object properties for accurate prefetching was not much used in current version but the work described in Chapter 3 and 5 could be used for object grouping which has been described by other researches. Some C++ programs are grouped manually to place sibling virtual functions in a single file. Objects in the same hierarchy are intentionally put in the file so that they can be located consecutively in the final executable file. This rearrangement have been done manually for large amounts of software but it can be automated by using the scheme shown in the thesis.

Object and page migration in distributed systems were suggested in the thesis. An implementation would give some realistic results to verify the feasibility of AP in distributed systems. Apart from other issues related to distributed systems, the future work would have to concern itself with the reduction of communication overheads for transferring objects or pages. In particular, the effect of AP in terms of overheads related to network software invocations are important parameters when evaluating its performance. AP could be tested on some object oriented distributed system such as SOS or COOL.

Another area of future research is investigating the effects of AP when it is adapted to a multiprocessor system. There are two issues for AP: one is fetching multiple objects or pages and the other is the coherence problem incurred by multiple fetches. A program running on a multiprocessor no longer has a single, sequential order of execution. The temporal and spatial locality, especially for shared data, of a processor is easily disturbed by actions of other processors. So, there should be a potential prefetching algorithm which includes the scheduling of pages to processors for optimum sharing. In the case of multiple objects or pages fetched to local memories in each processor, page coherence becomes more complicated. Li[Li 86] addressed the page coherence problem in a distributed shared memory system but assumed a single page fetch at a fault. The page coherence issue should be extended appropriately to multiple object fetches.

For shared memory multiprocessor system, shared data should not be in the same page so as to prevent reference contention. Using a page–oriented system, the programmer would optimize data reference patterns by laying out data structures and partitioning the work so as to make each node reference different sections of the linear address space. If two nodes write–share the same block of addresses, the virtual memory system will thrash[Chase 90]. These complicated problems cannot be resolved by natural or simple grouping algorithms but the AP approach could be a starting point for research because lookahead information in the prefetch table may give hints that could prevent the contentions.

Modern garbage collectors tend to get involved with the virtual memory hardware in order to speed the scan for garbage. Most object–oriented systems which support the persistent object model should have an algorithm for garbage collection algorithm. Suppose the persistent objects are listed in a prefetch tree, it is possible to do a short–term garbage collection by selecting objects which are unlikely to be referenced in the near future. This operation is based on a *distance* algorithm whereas the accurate prefetching is based on a *nearness* algorithm. Apart from this issue, the *distance* algorithm can be used to make practical Belady's MIN and VMIN algorithms.

Also a possible area is how to adapt hardware systems to AP. The time spent in looking up the prefetch table at every fault can be saved if it is carried out by hardware, e.g. an associative memory. Finally, the possibility of the applicability of AP to relational databases and other information systems that contain only implicit relationships in contrast to explicit pointers is not known. This area also needs further investigation.

References

[Abu-Sufah 81]

W.Abu-Sufah, D.J.Kuck, D.H.Lawrie, "On the Performance Enhancement of Paging Systems through Program analysis and transformations," IEEE Trans. Comp., vol. C-30, no.5, pp.341-356, May, 1981.

[Aho,86]

A.Aho, et.al. Compilers: Principles, and Tools, Addison Wesley, 1986.

[Baier 76]

J.L.Baier, G.R.Sager "Dynamic Improvement of Locality in Virtual Memory Systems," IEEE Trans. on Software Eng. Vol SE-2, No.1 March, 1976.

[Balter 90]

R.Balter, et.al, "Architecture and Implementation of Guide, an Object-Oriented Distributed Systems," IMAG Universities of Grenoble, Technical Report 1990.

[Barak 85]

A.Barak and A.Litman, "MOS: A Multiprocessor Distributed Operating System," Software Practice and Experience, vol.15 (8), pp 725-737, Aug., 1985.

[Baxter 89]

W.Baxter, H.Bauer, "The Program Dependence Graph and Vectorization," 16th ACM Symposium on Principles of Prog. Lang. Austin, Texas, 1989.

[Bergland 86]

E.J.Bergland, "An Introduction to the V-System," IEEE Micro, Aug., 1986.

[Breecher 89]

J.Breecher, "A Study of Program Restructuring in a Virtual Memory System," Performance Evaluation, 10, pp79-92, 1989.

[Brent 87]

G.A.Brent, "Using Program Structure to Achieve Prefetching for Cache Memories," Ph.D. dissertation, Univ. of Illinois at Urbana Champaign, 1987.

[Caceres 84]

Ramon Caceres, "Process Control in a Distributed Berkeley UNIX Environment," Report No. UCB/CSD/84/211. Dec., 1984.

[Campbell 88]

G.Jonston, R.Campbell, "A Multiprocessor Operating System Simulator," USENIX Proc. C++ Workshop Danver, Co, Oct., 1988.

[Cardelli 85]

L.Cardelli, P.Wegner, "On Understanding Types, Data Abstraction and Polymorphism," ACM Computing Surveys, Vol. 17, No. 4, Dec., 1885.

[Casavant 87]

T.L.Casavant, "Analysis of Three Dynamic Distributed Load Balancing Strategies with Varing Global Information Requirements," IEEE CH2439-81871, 1987.

[Chase 90]

J.S.Chase, et.al, "The Amber System: Parallel Programming on a Network of Multiprocessors," Proc. of the 12th ACM Symposium on Operating Systems Principles, 1989.

[Cheriton 88]

D.R.Cheriton, "The Unified Management of Memory in the V Distributed System," Standford Univ. Technical Report No. STAN-CS-88-1192, 1988.

[Courts 88]

R.Courts, "Improving Locality of Reference in a Garbage Collecting Memory Management System," CACM Vol.31, No.9, Sep., 1988.

[Cox 86]

Brad J. Cox, Object-Oriented Programming, An Evolutionary Approach, Addition Wesly, 1986.

[Dafni 87]

Dafni, "Texas Object-based systems," Ph.D. Dissertation, 1987.

[Dasgupta 90]

P.Dasgupta, et.al. "The Design and Implementation of the Clouds Distributed Operating System," USENIX Computing Systems, Vol.3, No.1, Winter 1990.

[Denning 79]

P.J.Denning, "Working Sets Past and Present," Tech. Rep. CSD-TR-276, Computer Science Department, Purdue Univ. May, 1979.

[Dewhurst 89]

S.C.Dewhurst, K.T.Stark, "Programming in C++," Prentice-Hall, 1989.

[Dewhurst 87]

S.C.Dewhurst, "The Architecture of a C++ Compiler," USENIX Proc. C++ Workshop Santa Fe. NM, 1987.

[Dixon 88]

G.N.Dixon, Object Management for Persistence and Recoverability, Ph.D. Thesis, The Univ. of Newcastle upon Tyne, 1988.

[Duglis 87]

F.Douglis, J.Ousterhout, "Process Migration, the Sprite Operating System," CH2439-8187, IEEE 1987.

[Edwards 64]

D.B.G.Edwards, "The ATLAS computing system," Information Processing Machines, pp43-55, Proc. in Prague on Sep. 7th-9th, 1964.

[Ezzat 86]

Ahmed K.Ezzat, "Load Balancing in NEST: A Network of Workstations," Proc. of Fall Joint Comp. Conf. pp1138-1149, 1986.

[Ferrari 76]

D.Ferrari, "The Improvement of Program Behavior," Computer, Nov. 1976.

[Finkel 86]

R.Finkel, et al, "Process Migration in Charlotte," Computer Science Tech. Rep. #655, Univ. of Winconsin–Dadison, Aug., 1986.

[Fleisch, 87]

B.D.Fleisch, "Distributed Shared Memory in a Loosely Coupled Distributed System," sigcomm 87, pp317–327, Aug. 1987.

[Gautron 87]

P.Gautron, M.Shapiro, "Two extensions to C++: A Dynamic Linkeditor and Inner Data," USENIX Proc. C++ Workshop Santa Fe. NM, 1987.

[Giraud 84]

F.A.Giraud, et al "Cache Prefetching with Chaining," IBM Tech. Disclos. Bull. pp2437, Vol. 27, Sept., 1984.

[Goldberg 83]

A. Goldberg and David Robson, Smalltalk–80, The Language and Its Implementation, Addison Wesly, 1983.

[Gorlen 87]

K.E.Gorlen, "An Object–Oriented Class Library for C++ Programs," Software–Practice and Experience, Vol. 17(12), Dec., 1987.

[Gornish 90]

E.H.Gornish, et.al. "Compiler–directed Data Prefetching in Multiprocessors with Memory Hierarchies," International Conf. on Supter Computing, ACM SIGARCH Comp. Arch. News, Vol.18, No.3, Sep., 1990.

[Habert 89]

S.Habert L.Mossseri, "COOL: Kernel Support for Object–Oriented Environments," Technical Report, LOO:EXP:REP:568, 1989.

[Hac 87]

A.Hac, X.Jin, "Dynamic Load Balancing in a Distributed System Using a Decentralized Algorithm," CH2439–8187, IEEE. 1987.

[Harland 87]

D.M.Harland, "OBJEKT: A Persistent Ojbect Store With An Integrated Garbage Collector," ACM SIGPLAN NOTICES V22, #4, Apr., 1987.

[Hatfield 71]

D.J.Hatfield and J.Gerald, "Program Restructuring for Virtual Memory," IBM Sys.J. 1971.

[Hartley 88]

S.J.Hartley, "Compile–Time Program Restructuring in Multiprogrammed Virtual Memory Systems," IEEE trans. on software eng., vol. 14, no.11, Nov., 1988.

[Hill 90]

M.Hill, J.Larus, "Cache considerations for multiprocessor programmers," CACM vol.33, No.8. Aug., 1990.

[Holliday 88]

M.A.Holliday, "Page Table Management in Local/Remote Architectures," International Conf. on Supercomputing, St.Malo, France, July, 1988.

[Horspool 87]

R.Nigel Horspool, Ronald M. Huberman, "Analysis and Development of Demand Prepaging Policies," The Journal of Systems and Software 7, pp183–194, 1987.

[Horwitz 88]

S.Horwitz et.al., "On the Adequacyy of Program Dependency Graphs for Representing Programs," Proc. of 155th ACM SIGACT–SIGPLAN Symposium on principles of Programming Languages, Jan., 1988.

[Hwang 84]

K.Hwang, F.A. Briggs, Computer Architecture and Parallel Processing, McGraw- Hill, 1984.

[Jorseph 70]

M.Joseph, "An Analysis of Paging and Program Behavior, Computer Journal 13, 48 - 54, 1970.

[Johnston 90]

G.M.Johnston, R.H.Campbell, "An Object–Oriented Implementation of Distributed Virtual Memory," An Internal Technical Paper, 1990.

[Jul 90]

E.Jul, N. Hutchinson, "Fine–Grained Mobility in the Emerald System," Object–Oriented Database Systems, ed. Zdonik SB and Maier, pp317–328, Morgan–Kuafman Pub. Inc. 1990.

[Kaehler 83]

T.Kaehler, G.Krasner, "LOOM -Large Object Oriented Memory for Smalltalk–80 Systems," pp251–270, Smalltalk–80 bits of history, words of advice. 1983.

[Kaehler 86]

T.Kaehler, "Virtual Memory on a Narrow Machine for an Object–Oriented Language," Prodeedings of the ACM Conf. on Object–Oriented Programming Systems, Languages and Applications, pp 87–106, Sep., 1986.

[Kahn 81]

K.C.Kahn, et.al. "iMAX: A Multiprocessor Operating System for an Object Based Computer," ACM 0–89791–062–1–12 /81–0127, 1981.

[Kaiser 88]

J.Kaiser, "MUTABOR, A Coprocessor Supporting Memory Management in an Object–Oriented Architecture," IEEE Micro, Oct. 1988.

[Kernighan 78]

B. Kernighan, D. Ritchie, "The C Programming Language," Prentice–Hall, 1978.

[Koenig 88a]

Andrew Koenig, "What is C+ + Anyway ?," Journal of Object–Oriented Programming, Vol.1, No.1, Apr/May, 1988.

[Koenig 88b]

Andrew Koenig, "An Example of Dynamic Binding in C+ +, JOOP, Aug/Sep., 1988.

[Lee 84]

J.K.F.Lee, "Branch Prediction Strategies and Branch Target Buffer Design," Computer Jan., 1984.

[Leffler 89]

S.J.Leffler, et.al, The Design and Implementation of the 4.3 BSD UNIX Operating System,Addison-Wesley 1989.

[Li 86]

K.Li, P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," 5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Calgary, Canada, 1986.

[Li 89a]

K.Li, R.Schhaefer, "A Hypercube Shared Virtual Memory System," 1989 International Conference on Parallel Processing, Aug. 8-12, 1989.

[Li 89b]

K.Li, P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," ACM trans. on computer systems, vol.7, no.4, pp321-359. Nov., 1989.

[Lippman 89]

S.B.Lippman, C+ + Primer, Addison-Wesley, 1989.

[Liskov 86]

B. Liskov, Guttag, J. Abstraction and Specification in Programming Development, ,MIT Press, McGraw-Hill, 1986.

[McCreary 89]

C.McCreary and H.Gill, "Automatic Determination of Grain Size for Efficient Parallel Processing," Vol.32, No.9, CACM Sept., 1989.

[McKusick 90]

M.K.McKusick, A "Pageable Memory Based Filesystem," UKUUG Summer '90, London, 9-13, July 1990.

[Meyer 88]

B.Meyer, "Object-Oriented Software Construction, Prentice-Hall 1988.

[Montenyohl 88]

M.Montenyohl, "Correct Flow Analysis in Continuation Semantics," Proc. of 15th SCM SIGACT-SIGPLAN Sympo. on Prog. Language, 1988.

[Nelson 87]

M.N.Nelson, et.al. "Caching in the Sprite Network File System," 1987, ACM Trans on Comp. System. Vol.6, No.1, Feb., 1988.

[Organick 72]

E.I.Organick, "The Multics System: An Examination of Its Structure, Cambridge, Mass.: M.I.T.Press, 1972.

[Parrington 88]

G.D.Parrington, Management of Concurrency in a Reliable Object-Oriented Computer System, Ph.D. Thesis, The Univ. of Newcastle upon Tyne, 1988.

[Popek 85]

G.J.Popek, B.J.Walker, "The LOCUS Distributed System Architecture," The MIT Press 1985.

[Pyster 88]

A.B.Pyster, "Compiler Design and Construction, Tools and Techniques," VAN Nostrand Reinhold Company, 1988.

[Renesse 89]

R. Renesse, A.S.Tanenbaum, A.Wilschut, "The Design of a High-Performance File Server," pp22-27 The 9th Distributed Computing Systems, Newport Beach, California, June5-9, 1989.

[Ruggieri 88]

C.Ruggieri, T.Mrtagh, "Lifetime Analysis of Dynamically Allocated Objects," Proc. of 15th SCM SIGACT-SIGPLAN Sympo. on Prog. Language, 1988.

[Scheurich, 88]

C.Scheurich, M.Dubois, "Dynamic Page Migration in Multiprocessors with Distributed Global Memory," IEEE Trans. on Computers, pp1154-1163, Aug., 1989.

[Seltzer 90]

M. Seltzer, et.al, "Disk Scheduling Revisited," USENIX Conference Proceeding, Winter 1990.

[Shapiro 89]

M.Shapiro, et.al., "SOS: An Object-Oriented Operating System - Assessment and Perspectives," Computing Systems, Vol.2, No.4, Fall 1989.

[Silberschatz 88]

A.Silberschatz, J.L.Peterson, "Operating System Concepts," Addison-Wesley, 1988.

[Smith 78]

A.J.Smith, "Sequential Program Prefetching in Memory Hierarchies, IEEE Computer 11, 7 - 21, Dec., 1978.

[Smith 88]

A.J.Smith, "Cache Memory Design: An evolving art," IEEE Spectrum, Dec., 1987.

[Smith 81]

J.E.Smith, "A Study of Branch Prediction Strategies," proc. of Computer Architecture, 1981.

[So. 88]

K.So, R.N.Rechtschaffen, "Cache Operations by MRU Change," IEEE Trans. on Computers, Vol.37, No.6, June, 1988.

[Sollins 79]

K.R.Sollins, "Copying Complex Structures in a Distributed System," Ms.C Thesis, MIT. 1979.

[Stamos 85]

J.W.Stamos, "Static Grouping of Small Objects to Enhence Performance of a Paged Virtual Memory," ACM Trans. on Computer Systems, Vol.2, No.2, May, 1984.

[Stroustrup 86]

B.Stroustrup, The C+ + Programming language, Addison Wesly, 1986.

[Stroustrup 87]
B.Stroustrup, "Possible Directions for C+ +," Proc USENIX C+ + workshop, Nov., 1987.

[Swinehart 86]
D.Swinehart, P.Zellweger, R.Beach, R.Hagmann, "A Structural View of the CEDAR Programming Environment," Transactions on Programming Languages and Systems, Vol.8, No.4, pp 419–490, Oct., 1986.

[Synder 78]
R.Synder, "On the application of a priori knowledge of program structure to the performance of virtual memory computer systems," Ph.D. dissertation, Univ. Washington, Nov., 1978.

[Taenzer 89]
David Taenzer, "Problems in Object-Oriented Software Reuse," ECOOP 89 Proceedings, pp23–pp38, 1989.

[Tetzlaff 89]
W.H.Tetzlaff, M.G.Kienzle, and J.A.Garay, "Analysis of block-paging strategies," IBM Jour. of Research and Development, Vol.33, No.1, Jan., 1989.

[Theimer 85]
M.M.Theimer, K.A.Lantz, D.R.Cheriton, "Preemptable Remote Execution Facilities for the V-System," ACM Operating Systems Review, Vol 19, No.5, Dec., 1985.

[Tieman 89]
M.Tieman, "G+ + 1.34.1 Manual," 1989.

[Thomasian 87]
A. Thomasian, "A Performance Study of Dynamic Load Balancing in Distributed Systems," IEEE, CH2439-8187, 1987.

[Verhoff 71]
E.W.Verhoff, "Automatic Program Segmentation Based on Boolean Connectivity," SJCC 1971.

[Williams 87]
I.W.Williams, et.al., "Realization of a Dynamically Grouped Object-Oriented Virtual Memory Hierarchy," Proc. of APPINS, 1987.

[Young 87]
M.Young, R.Rashid, et.al. "The duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," CMU Tech.Rep.15, Feb., 1987.

[Zayas 87]
E.R.Zayas, "Attacking the Process Migration Bottleneck," Proc. of the Eleventh ACM Sympo. on Operating Systems Principles, Austin, TX, pp13–24, ACM Operating System Review 21, 5. Nov., 1987.