Algorithms for Object Pre-fetching in a Distributed Persistent Object System

by

Jun Chen

A thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

ALGORITHMS FOR OBJECT PRE-FETCHING IN A DISTRIBUTED PERSISTENT OBJECT SYSTEM

BY

JUN CHEN

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

Over the past decades, microprocessor speed has dramatically improved whereas memory access time has improved significantly less. Thus, an important factor in achieving good program performance is now ensuring that data can be loaded from memory efficiently (hence the proliferation of multiple levels of cache memory). In large-scale distributed systems, program execution time depends not only on the memory access latency but also on the network latency. Thus, in such systems, it is critical that data can be loaded from remote machine efficiently. This is particularly true in systems where data is shipped between machines. Included among such systems are certain distributed persistent object systems (all of which provide transparent distributed access to objects whose state automatically persists across object activations and power failures).

One of the solutions used to narrow the expanding gap between microprocessor and memory speed is the use of data pre-fetching where a prediction is made of what data will be needed next and that data is pre-loaded into the cache before it is needed. Extensive research has been done on the application of pre-fetching in a number of ways to decrease effective memory latency. Less work, however, has been done on pre-fetching in distributed systems (with the limited exception of distributed database systems) and the available literature related to the design and implementation of algorithms for *object* pre-fetching in a persistent distributed object system is extremely limited. Despite this, pre-fetching objects offers the promise of significant reductions in observed network latency in such systems. Further, object systems are semantically rich and provide a wealth of inter-object relationships that could be exploited as the basis for pre-fetching strategies.

In this thesis, a family of strategies for object pre-fetching in a distributed shared virtual memory (DSVM) system is proposed. All the strategies are based on using a per-object "reference predictor" that is associated with an object's entry in the Global Directory of Objects (GDO) thereby providing object-specific pre-fetching ability. Each such reference predictor records some history of object access patterns based on recent method invocations made by the corresponding object. This information is then used to predict which objects should be pre-fetched when a given object is active. After defining a basic predictor, the concepts of pre-fetching "threshold", pre-fetching "depth" and "path-based pre-fetching" are introduced to improve both pre-fetching efficiency and accuracy. Finally, the data structures and algorithms required to implement six specific pre-fetching strategies are presented and their applicability is discussed.

Acknowledgements

I am deeply grateful to my supervisor, Dr. Peter Graham, for his guidance and encouragement during my thesis work. I also wish to thank the members of my thesis committee: Dr. Sylvanus Ehikioya and Dr. Bob McLeod for their valuable suggestions on the draft of this thesis. Finally, I would also like to express my sincere gratitude to my wife and parents for their love, understanding, and encouragement without which this research would not have been completed.

Table of Contents

List of Figures

1. Introduction

The technology advances in high-speed processors have significantly outpaced the corresponding advances in memory systems. Over the past few decades, microprocessor speed has improved at 50-80% per year while memory access time has only improved by 5-10% per year [2]. This situation has required developers to design highly efficient techniques to narrow the expanding gap between the speed of microprocessors and memory access. This problem has been solved primarily through the addition of hardware caches between the CPU and main memory, which provide faster access to copies of recently used data. When data is referenced, if it is not found in the cache, then it is copied into the cache so it will be there next time it is needed. Since programs exhibit a high degree of locality in the data they access, this tends to result in useful data frequently being found in the cache (a "hit"). The entire "caching" process is transparent to the machine's users/programmers. Caches are typically organized as a number of equal-sized lines/blocks which store recently used portions of data from the main memory. The CPU can access the data in the cache at near to processor speeds[1].

Using caches, however, will always result in a cache miss (i.e. desired data not found in the cache) the first time a program tries to access a given data block because only previously accessed data can be in the cache (this is referred to as a "cold start" or "compulsory miss"). This is also true for the first access to remote data in a distributed system. Further, in distributed systems, accesses to the same shared data by processes at multiple sites may cause that data to move between sites so that many compulsory misses may occur at each site. Finally, the penalty for misses (i.e. the latency required to load data remotely) is extremely high in distributed systems. All these factors suggest that something more than simple caching is required.

If data could be loaded quickly (either into a cache or into memory, in the case of a distributed system) there would be no problem. Since this is not possible, however, some

[1] This is true for the first level (L1) cache. Subsequent levels of cache (L2 and L3) are increasingly slower than the CPU but are still faster than main memory.

mechanism for decreasing the effective latency of such loading must be found. One effective way to improve this situation is by the use of data pre-fetching [28]. Pre-fetching is a technique that attempts to eliminate cache misses by predicting the data (i.e. memory blocks in a hardware cache) that will be referenced in the near future and bringing them into the cache before they are actually needed. Ideally, such pre-fetching would complete just in time for the processor to access the needed data thereby avoiding any delays in processing and also ensuring that other, potentially useful data is not replaced in the cache (to make room for the pre-fetched data) until absolutely necessary. Various techniques have been proposed for pre-fetching in hardware caches [11][14][15][53][57] and some for pre-fetching in distributed systems [8][12][23][38][48] as well.

Object-oriented programming has become the de-facto standard for many software development problems and, as a result, object systems are becoming increasingly commonplace. All object systems have certain common properties that distinguish them from non object-oriented software systems. For example, in object systems, each object has a unique object identifier (OID) that distinguishes it from other objects. Each object also encapsulates its data and has a set of functions, usually called methods, that manipulate the data. An object communicates with other objects by making method invocations on them. Further, objects are instantiated from classes that define their characteristics and similar classes are related to one another via inheritance. Thus, a specialization of one class (e.g. student is a specialization of person) may inherit the properties of its parent class. These characteristics offer benefits in terms of reusability and maintainability of software, in addition to simplification of software design. As will be shown (see Chapter 4), these object characteristics can also be exploited to enable effective object pre-fetching.

Adding persistence and transparent distribution to object systems offers significant benefits which include a simplified programming environment and improved sharing. Such benefits have contributed greatly to the growing acceptance of distributed object systems as an architecture for large-scale software development[2]. The question then arises, whether or not pre-fetching can be effectively used in distributed object systems

[2] Although the addition of transparent persistence has lagged somewhat.

where objects are moved from machine to machine in response to method invocations made on them (i.e. data and function/object shipping as opposed to call shipping as with remote method invocation). This topic is the focus of the thesis.

Despite a significant amount of literature on memory pre-fetching (for hardware caches), and data pre-fetching (both for database/file system applications and, to a lesser extent, distributed applications)), data structures and algorithms for the pre-fetching of *persistent objects* have been rarely considered. In a large-scale distributed system with many processors interconnected via a network, thousands of persistent objects may exist in the system and interact with each other. The program execution time depends significantly on the network latency for moving objects from remote machines to another machine's local memory. Thus, for such distributed systems, the performance gap between processor speed and object access speed (now dominated by the network latency) is exacerbated and efficient pre-fetching strategies are of great importance to hide the latency of remote object access.

Some strategies for object pre-fetching, mostly in the context of object database systems, have been previously proposed (e.g. [8][12]). These techniques exploit object semantics (e.g. inter-object calling relationships) to attempt to cluster related objects together into a single page. In this way, when one object is referenced, its entire page is loaded and thus, additional, useful objects are pre-loaded with it. This is a form of pre-fetching where the determination of what should be pre-fetched is done early in the life of the system. This saves overhead at run time but may decrease accuracy if object relationships change.

In this thesis, the design of a number of "reference predictors" is proposed. Each such predictor will record inter-object access patterns (determined by the method invocations made by each object), analyze the recorded information using a variety of proposed algorithms and thereby predict a set of one or more objects that are likely to be accessed in the near future and which, therefore, should be pre-fetched. Each reference predictor's data structures will be associated with the entry in the global directory of objects (GDO) for the object to which they apply and prediction services will be provided through the GDO. Thus, the GDO is not only a repository for information about the objects being maintained in the system, but is also a provider of management services on

those objects. To provide efficient object pre-fetching, several pre-fetching strategies will be presented. These include: pre-fetching "threshold", pre-fetching "depth" and path-based pre-fetching. The data structures and algorithms required to collect and use the inter-object access information (and thereby to implement the reference predictors) for six different pre-fetching strategies that use various combinations of pre-fetching threshold, depth, and paths will be described. Although these techniques are designed in a specific distributed persistent object system – DSVM (Distributed Shared Virtual Memory System), they are applicable to other persistent object systems.

1.1. Organization

The remainder of the thesis is organized as follows. Chapter 2 gives an overview of object systems and various pre-fetching techniques used in other environments. Chapter 3 presents the key features of the assumed distributed persistent object environment and discusses the general problem of predicting which objects to pre-fetch. In Chapter 4 the design and implementation of the data structures and algorithms used for object pre-fetching based on historical method invocation patterns are discussed. Six specific strategies for object prediction are described in detail. Chapter 5 speculates on the likely applicability of each of the six strategies and considers trade-offs between them. Finally, Chapter 6 makes some concluding comments and suggests directions for future research.

2. Background and Related Work

Before presenting the research problem (see Chapter 3) an overview of object systems, the assumed execution environment for this thesis, pre-fetching techniques (as used in various areas) and branch prediction are provided. This overview will provide background information and introduce key concepts in subject areas that are relevant to the thesis.

2.1. Object Systems

Object-oriented (OO) concepts first emerged in the 1960s' in the context of the language Simula 67 [13]. Object orientation offers many advantages and, over the last 15 years, with the growing availability of various object-oriented languages and supporting environments as well as the existence of more powerful hardware, the object-oriented paradigm has come to have a significant influence on many areas of computer science. A system based on an OO approach is viewed as a collection of objects with characteristics/features including class, inheritance, encapsulation and polymorphism [4][34][42]. Objects interact with each other via messages that contain information used in invoking operations on the appropriate objects. Systems built using OO techniques are typically more robust, extendible, reusable and maintainable than traditionally structured systems and thus the OO paradigm is very attractive[3][6].

2.1.1. Object Characteristics

An object refers to a run-time instance of something that represents a real-world entity. Each object contains both structural and behavioral components.

The structural components of each object include an Object Identifier (OID) and the object's attributes (data representing the state of the object). In OO systems, each object is distinguished from other objects by having its own unique identifier. The identity of an object is independent of conduct, type and addressability. This means that the identity of an object remains permanently associated with that particular object despite any structural or behavioural changes. The ability to uniquely identify each object is essential

for managing large collections of objects and maintaining relationships among them. A set of values for the attributes (data) defines the state of each such uniquely identified object. In OO systems, objects are, in a general sense, similar to tuples in a relational database system.

The behavioral component of each object consists of a set of functions, usually called methods, that manipulate the attributes of the object. They are the only way of accessing the structural component and hence, of modifying the internal state of an object. An object invokes an operation in another object by communicating with it using a message. Note that messages are not the same as methods. Messages are used for communications between objects, while methods respond to messages and act on the attributes of an object.

Encapsulation means that a class' internal implementation is hidden from the outside world. The outside world sees only an external view consisting of available services and properties of these services (sometimes referred to as the class' interface). Using this technique, an object's internal format is insulated from other objects. This ensures that only an object's methods can be used to access the object's attributes rather than allowing direct access by other objects. This has the effect of protecting an object's state and limits the scope of effect when changes are made thereby making it easier to isolate code bugs. Encapsulation also serves to improve the reliability and understandability of OO software.

Similar objects in a system can be classified according to their behavioral and structural components. A class (or type) is a set of objects that have the same characteristics in terms of structure and behavior. The relationship between an object and its class is called the "instance-of" or "part-of" relationship.

Reusability of software has long been an important goal in programming language design. The concept of class provides a good modular decomposition technique that provides a basis for the reuse of object definitions. A new class may be built as an extension of an existing class by adding new attributes and/or methods to it. This process is known as inheritance. The new class is called a subclass of the existing class and the existing class is referred to as a superclass of the new class. In terms of the structural and

behavioral components, the objects in the subclass inherit the attributes and methods from its superclass. The term inheritance is borrowed from the biological concept of inheritance in which traits are inherited from parents, who themselves inherit their traits from their parents. A significant benefit of class inheritance is the ability to adjust object structure and/or behavior to new requirements by creating new inherited classes without impacting the existing classes.

Polymorphism is another important characteristic in OO systems. Using polymorphism, different classes of objects can interpret messages in their own ways. Thus, the same message applied to objects from different classes can result in different behaviors. More importantly, polymorphism permits objects from related classes to be treated in a uniform way, which leads to more general and re-usable programs.

2.1.2. Persistent Object Systems

Persistence is the property of an object "through which its existence transcends time (i.e. the object continues to exist after its creator exits) and /or space (i.e. the object's location can move from the address space in which it was created)" [6]. An object in memory can be made persistent by ensuring that it is written to disk when required so that the object can later be retrieved by other applications. The object will have the same state and relationship to other objects when retrieved, as at the time it was last accessed. The lifetime of a persistent object can, however, be explicitly terminated after which its state becomes inaccessible and its persistent storage is reclaimed. An object's identity, however, is immutable (i.e. never reused) even after the object is deleted [2]. There are several advantages of persistent systems [43]. First, it is easy for the system to recover previous run-time state after power or system failures. Second, persistent systems reduce complexity for application developers because they need not write I/O code to explicitly save state to disk or retrieve it from disk. Third, they provide the potential to reduce code size and execution time. Fourth, since all data resides in one persistent store, a single uniform model of protection may be employed.

Persistent object systems, of course, also provide OO features. In contrast to traditional relational databases, which represent data in the limited form of tables/relations, OO systems offer flexibility in data manipulation through object

semantic links (i.e. classification, inheritance, polymorphism and encapsulation). Combining the features of persistence and objects leads to a powerful programming environment.

2.1.3. Distributed Object Systems

In a distributed system, a collection of data is placed in memory and/or on disk storage at different sites across the network where they can be accessed or shared using the network by a number of applications whose components reside and execute at (potentially) multiple different sites in the network. A local area network (LAN) provides data access to applications within a localized environment (e.g. a University department), while a wide area network (WAN) provides data access to a number of geographically dispersed sites. Distributed systems may be localized (e.g. NFS within the University) or broader in scope (e.g. the domain name service/DNS used in the Internet). Generally, distributed systems provide better flexibility and scalability than centralized systems. For example, if an organization grows by adding new branches at different cities, distributed systems can support smooth incremental modifications to their IT systems with minimal impact on existing software. These advantages, however, come at a cost – increased complexity in design and implementation. In part in response to this complexity, the trend in distributed systems has evolved towards distributed objects.

The concepts of distribution and object orientation can be combined to build powerful distributed OO systems. There have been several general purpose, research-oriented distributed object systems reported including: ITASCA [24], ENCORE [23], GOBLIN [27], THOR [38] and EOS [46]. Many special-purpose distributed object systems have also been deployed in industry. Distributed computing, in general and object distributed computing specifically are rapidly emerging as the defining architectures for the entire computer industry [44].

Remote Procedure Call (RPC) and standards such as the distributed computing environment (DCE) [44] have been created to enable communication between distributed processes in non object-oriented distributed systems. Several groups have also proposed and developed mechanisms to allow objects in distributed systems to communicate. For example, Sun Microsystems has added a feature, Remote Method Invocation (RMI), to

the Java Language to enable distributed inter-object messaging. RMI allows one to create Java objects whose methods can be invoked by Java code running on a Java virtual machine on a different computer. Using RMI, Java objects residing on a server do not have to be downloaded to a client for execution as was originally required. Further, objects are not required to reside on a single server. This approach is in direct contrast to the object shipping technique considered in this thesis.

The Object Management Group (OMG) has made the Common Object Request Broker Architecture (CORBA) a defined standard. CORBA combines object technology with a client-server model to provide a uniform view of an enterprise's entire computing system. By inserting an Object Request Broker (ORB) between the client and server components, CORBA lets each object reside anywhere in the system and communicate with each other using their names through the broker. It also provides persistence, events, transactions, querying and properties services through the ORB interface.

More recently, other systems such as SOAP (Simple Object Access Protocol) [18][10] and XML-RPC (Extensible Markup Language-Remote Procedure Calling protocol) [62] have been developed which also provide support for developing advanced distributed applications. SOAP is a lightweight protocol for exchanging information between objects in a distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP makes distributed computing possible in a multiplicity of forms using XML as a communication medium and thereby simplifies the complexities of cross-platform and cross-language interaction. XML-RPC is a Remote Procedure Calling protocol that allows software running on distributed systems, running in different operating environments to make procedure calls over the Internet. Its remote procedure calling uses HTTP as the transport and XML for data encoding. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

2.1.4. Distributed Shared Virtual Memory (DSVM)

In contrast to RMI, CORBA and SOAP, some systems move objects from machine to machine in order to allow method invocations to execute locally (instead of executing methods remotely). This may be done to enhance security, to provide implicit dynamic load balancing (objects migrate to where they are needed and the computation is done there) and/or to enable simplified programming practices (since the complexity of remote invocations may be hidden by transparent object migration).

Interest has been expressed recently in both transparent persistence and transparent distribution for object programming. Graham, et al. [22][51] proposed a distributed shared virtual memory system (DSVMs) that provides a uniform view of a persistent object space in a single shared virtual memory distributed across multiple interconnected machines. Their environment will be the one assumed in this work.

Exploiting the availability of 64 bit address machines, their work seeks to build a distributed persistent object programming environment in a single, 64 bit, shared address space. Objects placed in this persistent memory can be shared and are accessible (via transparent migration from site to site) across a collection of interconnected machines using extended Distributed Shared Memory (DSM) [25] techniques. Building a persistent object system in this way offers a number of advantages including: simplified programming, improved sharing and potentially improved performance. Specific implementation benefits include eliminating the need to "swizzle" [26][58][61] object references (i.e. dynamically re-map an object's OID to its in-memory address). This is because, in a 64-bit DSVM system, a persistent object's virtual address can be directly used as its OID (i.e. object reference) [40]

Figure 2.1 shows the high level structure of the proposed distributed persistent object system. Every processing node in the system is a fully functional computer containing a processor (capable of addressing the entire virtual address space), a large physical memory, and a local swap disk. The nodes are interconnected by a high-bandwidth, low-latency network (e.g. Myrinet, SCI, or Gigabit Ethernet). Objects that are distributed across the computers collectively implement a number of different applications executing over the network. A LAN provides object access to these

applications and, with the right consistency protocol, a WAN could even, conceivably, provide object access to a large number of geographically dispersed sites.

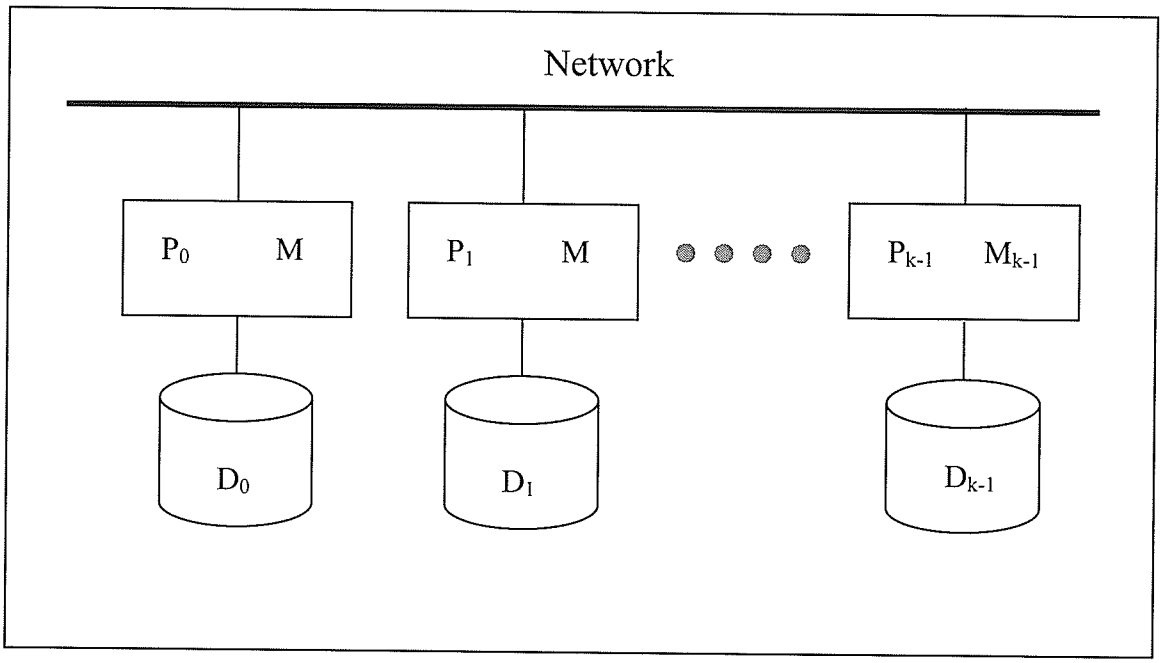

Figure 2.1 - High Level DSVM Architecture

2.2. Pre-Fetching Techniques

Over the past few decades, increases in microprocessor clock speed and the availability of pipelined and superscalar processor implementations have dramatically improved microprocessor execution speeds. Unfortunately, while microprocessor speeds have been improved by 50%-80% per year, memory access time has only improved by 5%-10% per year. This situation has required computer designers to develop new techniques to narrow the expanding gap between microprocessor and memory speeds. To address this problem, multiple levels of caches have been inserted between the CPU and main memory.

Caches are small, fast memories that are typically organized as a number of equally sized "lines" each of which store a copy of some recently used portion of main memory. The CPU is then able to access data in the cache(s) much faster than it can access data in main memory. As long as the needed data is found in the cache, performance is increased. The performance of a cache memory system can be tuned by optimizing

certain cache parameters including: cache size, cache line size, placement policy (what data goes where in the cache), replacement policy (which data are removed when the cache is full) as well as the number of levels of cache.

While caches work very well for many types of processing, there are certain programs for which caches provide only limited benefits.[3] Given the growing importance of cache effectiveness in determining overall system performance, this is a serious issue. One way to address this problem is through a technique known as "pre-fetching". Pre-fetching attempts to eliminate unnecessary cache misses (when required data is not found in the cache) by predicting the memory blocks that will be referenced in the near future and bringing them into the cache just before the processor needs them (i.e. "pre-fetching" those blocks). Ideally, pre-fetching will load the required blocks just in time for the processor to access the needed data thereby both avoiding stall cycles as well as ensuring that no data is replaced in the cache until absolutely necessary. Pre-fetching of data blocks into the cache can be initiated using either hardware or software-based schemes. While this thesis is focused on the pre-fetching of objects in distributed systems, it is worthwhile reviewing the work on pre-fetching done for hardware caches.

2.2.1. Software Pre-fetching Schemes

Usually, software pre-fetching is done based on compiler analysis and code modification. The compiler attempts to predict which memory accesses have the possibility to cause cache misses and avoid their occurrence by generating instructions to pre-fetch the needed memory blocks into the cache before they are needed [45][21][7]. Software pre-fetching is often used for scientific code that uses many loops to perform large array calculations. Within such loops, the memory reference patterns that the program will generate are often predictable and provide excellent information to use in generating pre-fetch code. The main advantage of software pre-fetching is that it is highly accurate when access patterns can be determined (since the compiler knows the access patterns before inserting the pre-fetch instructions into the program). It is also less likely that unneeded data will be brought into the cache unnecessarily (leading to "cache pollution").

---

[3] This is because caches depend on programs exhibiting good "locality of reference" (frequent re-use of recently accessed or nearby data) and not all programs do so.

Software pre-fetching, however, also has some disadvantages, which include:

- code expansion and the resulting potential increases in execution time (due to decreased locality of reference and the cost of executing the pre-fetch code),
- lack of flexibility (to handle changing pre-fetch requirements), and
- the difficulty of inserting pre-fetch code at the appropriate place.

Since pre-fetch instructions have to be embedded into the program, the code size increases. If there are many pre-fetch instructions inserted, the code will grow significantly. The execution time required for the pre-fetch instructions will also result in some increase in the total execution time of the program. Making pre-fetch decisions at compile time also assumes a particular memory reference pattern. If the predicted referencing pattern is wrong, a high performance penalty may result. In dynamic program environments, where the pre-fetching requirements may change unexpectedly, this is a serious problem. Finally, a performance penalty will also be incurred if the compiler schedules pre-fetch instructions at unfortunate times. If the pre-fetch is too early, the cache may evict useful data to make room for the new pre-fetched data that is not yet useful (this is referred to as cache pollution). If the pre-fetch is too late, the data will not be in the cache when the processor needs it. Therefore, deciding exactly where to insert the pre-fetching code in the program is a crucial issue for the compiler.

2.2.2. Hardware Pre-fetching Schemes

Hardware pre-fetching is a scheme where pre-fetching activities are determined by recent program execution behavior using hardware and without explicit involvement by the compiler. Hardware pre-fetching relies on speculation about future memory access patterns based on (recent) previous access patterns. There have been two main approaches proposed [14][15] for hardware pre-fetching: sequential pre-fetch and stride pre-fetch.

Sequential pre-fetch:

In this scheme, pre-fetching exploits spatial locality in data access patterns and pre-fetches consecutive memory blocks [55][57]. The "One block Lookahead Scheme" (OBL) is the simplest form of sequential pre-fetch in which the next block, b+1, is pre-

fetched when the current block, b, is referenced. There are three schemes that have been derived from OBL. In the "always pre-fetch" scheme, the next block b+1 is always pre-fetched when block b is accessed. This scheme should decrease cache misses but cache pollution could be a problem since many unnecessary blocks might be pre-fetched. In the "pre-fetch on miss" scheme, the next block, b+1, is pre-fetched whenever the access to the current block, b, results in a cache miss. This scheme can reduce cache misses by up to 50% [54] and the demand on cache resources due to the pre-fetching is lower than that of the always pre-fetch scheme. In the "tagged pre-fetch" scheme, every cache block is associated with a tag bit and whether or not the next block is pre-fetched is determined by the state of the tag. Only when a fetched block or a pre-fetched block is referenced for the first time, is the tag bit set and pre-fetching for the next memory block is initiated. Tagged pre-fetching is more expensive to implement because of the addition of the tag bits to each cache line and because a more complex cache controller design is required. It should, however, outperform the other strategies.

To improve pre-fetch efficiency, one can take OBL a step further by pre-fetching several consecutive blocks following the missed block in the cache on each cache miss [14][5][57][53]. In this scheme, when the current block b is accessed, the next few blocks b+1, b+2, b+3, ... , b+k will be pre-fetched instead of just 1 block. The value of k is known as the degree of pre-fetching and determines the number of blocks to be pre-fetched [57].

Stride Pre-fetch:

When blocks that need to be pre-fetched reside in non-consecutive memory blocks, sequential pre-fetch will cause ineffective pre-fetching and that might result in cache pollution and unnecessary cache misses. The "stride pre-fetch" scheme [11][15] addresses this issue. The algorithm calculates the difference between the address of the current and previous memory blocks and uses this difference (the "stride") to predict the address of blocks that should be pre-fetched in the future. For example, when the current address 'b' is accessed, the stride between the current address 'b' and previous address 'a' is computed as 's=b-a'. Assuming the stride is constant during program accesses, the next address 'c' should be predicted to be the sum of 'b+s' and this block is the one that will be pre-fetched. To implement this scheme, a reference prediction table [11] or a

stride prediction table [15] is used to hold data addresses that have been recently used by memory access instructions together with the pre-fetch strides calculated for those data addresses.

Generally, the advantages of hardware pre-fetching include flexibility and its greater suitability for general applications. Since the prediction is based only on recent behavior, however, hardware pre-fetching may not always be as accurate as software-based pre-fetching. In such cases, this may result in inaccurate predictions that could lead to cache pollution and resulting cache misses.

2.2.3. Work Related to Distributed Object-Based Pre-fetching

In a distributed object-based environment, there are two important things that should be considered when developing pre-fetch schemes: network latency and inter-object characteristics. In distributed systems, computers at different sites communicate with each other through a network. Despite significant improvements in the bandwidth of networks, many common network technologies (including the most prevalent one – Ethernet) are still limited in terms of the speed with which a data transfer from one node to another may be initiated. This is the so-called startup "latency" of communications. To hide costly round trip times on the network and thereby reduce memory access times, pre-fetch schemes could be used to bring data from remote nodes to local accessing nodes before it is actually needed by the local processors. In object-based systems, inter-object relationships provide important information on object access patterns that can be exploited by pre-fetch schemes to predict likely future object accesses.

Three general pre-fetching strategies have been proposed[4] based on how candidate objects for pre-fetching are selected. The first type of pre-fetching strategy is referred to as "aggressive" or "selective eager" pre-fetching [23][1], where all objects from a stored page or segment are extracted together when the first object from that page or segment is fetched. These schemes are based on the assumption that objects in the pages are well clustered and they do not use any sort of advanced object semantics to predict future object accesses. Such eager pre-fetch schemes improve object-hit rates but may lead to

[4] These strategies have been proposed for use in various database system environments and typically do not consider distribution.

many page faults (or communications if applied in a distributed environment) as well as unnecessary copying overhead when clustering does not match actual object usage patterns.

In Thor [38], a simple sub-segment pre-fetching policy is used to pre-fetch groups of objects instead of the contents of pages or segments. Each segment is split into groups containing k objects each. In response to a fetch request for a particular object, the group containing the object is pre-fetched. The size of groups in each segment can be adjusted dynamically to achieve the best possible performance. When the clustering of objects into segments matches the application access pattern, the scheme quickly increases the pre-fetch group size until a large fraction of a segment is sent in response to each fetch request, thereby reducing the number of network round-trips. When clustering and the application access pattern do not match very well, the dynamic scheme quickly reduces the pre-fetch group size until only a small number of objects are sent in response to each fetch request. This reduces the number of useless objects sent and thereby avoids removing useful objects from memory to make room for useless ones.

The second type of pre-fetching strategy exploits object semantics such as inheritance and structural inter-object relationships to predict likely future access patterns. Chang and Katz [8] proposed a run-time clustering and buffering algorithm using user-defined hints which determined which type of object semantics (e.g. configuration relationships, version history or correspondence) should be used. At the beginning of an interaction with the object base (which is their assumed execution environment), the users provide the buffer manager with a single hint to be used. Through this hint, accessing an object causes the page containing it and the pages containing its immediate component objects to be brought into the buffer pool. This achieves extremely good performance when applications execute along a certain hint.

Cheng, et al. [12] extended this work by adding multiple hints, pre-fetch depth and physical storage considerations. They proposed a profile-based buffering scheme to customize the pre-fetching and replacement policies for each individual client. The implementation of the profile is based on relationships among objects and an application program is designed to traverse the semantic links according to some meaningful patterns. Instead of using a single hint, a series of hints are given for all types of

relationships. A pre-fetching depth was also added to each user hint according to the semantics of the corresponding relationship. In their proposal, pre-fetching depths are related to the level of object clustering. Level-one clustered (primarily clustered) objects are grouped into the same or adjacent disk page, while level-two and higher-level clustered (secondarily clustered) objects are stored on the same or adjacent tracks/cylinders but not necessarily on the same page. Objects clustered at each level belong to the corresponding pre-fetching depth. Thus, for example, given a pre-fetching depth of 1, objects clustered in the same page would be pre-fetched.

Knafla [31] also presented an approach to predict page accesses by using relationships between objects in a client-server environment. A discrete-time Markov Chain [35] is used to model the relationships between persistent objects and a technique called "hitting times" (i.e. the mean time needed to traverse from a current object to an object in another page) is used, in part, to compute the page access probability. The computed probability is then compared to a threshold defined by certain cost/benefit parameters. If the probability of a page is higher than the threshold, then the page is a candidate for pre-fetching.

The third type of pre-fetch strategy predicts likely-to-be-used objects by maintaining a history of past object access patterns. Palmer and Zdonik [48] propose a predictive cache that employs an associative memory to recognize each access pattern individually and make prediction decisions within each access context according to the history of that context.

2.2.4. Related Work in Branch Prediction

The concept of predicting behavior (both for pre-fetching and for other applications) has been used in a variety of areas in computer science including microprocessor design [58], virtual memory paging [46], file systems [36], the WWW [63], and databases [18]. The goal of all predictors is to obtain high prediction accuracy while minimizing the overhead of the prediction decision. Much of the experience gained in doing prediction in other environments (e.g. database systems) is applicable to the problem of pre-fetching objects. In particular, some ideas used in branch prediction are directly applicable. Thus, the basics of branch prediction are now reviewed.

Conditional branches are one of the most serious impediments to high performance in modern processors using superscalar or super pipelined designs. Branch prediction schemes try to eliminate the impact of conditional branches by predicting the outcome of the branch instructions at the instruction fetch stage of the pipeline and issuing subsequent instructions before the actual outcome is known. Branch prediction can be accomplished in one of three ways: using static prediction done at compile-time via compiler analysis, using dynamic prediction at run-time exploiting special hardware structures or using hybrid prediction which combines both static and dynamic prediction techniques to attempt to improve the accuracy of prediction.

Static Branch Prediction:

Static branch prediction is done in software using information about the program that is known to the compiler. This can be as simple as predicting that all branches are not taken or that all branches are taken [35] or always predicting that forward branches are not taken and backward branches (commonly implementing loops) are taken [54][50]. In this latter case simple program semantics are being used to predict the most likely control flow path to be followed. Unfortunately, accuracy is limited in certain, important cases since the semantic analysis is incomplete. McFarling, et al. [41] used static profiling information to predict branch paths by measuring the tendencies of the individual branches. The main limitation of this approach is that program profiling has to be performed in advance using certain sample data sets, which may turn out to have different branch tendencies than the data sets that actually occur at run-time.

Dynamic Branch Prediction:

Dynamic branch prediction takes advantage of knowledge of branches' run-time behavior to make predictions. Schemes such as the "Branch History Table" (BHT) [25] are classified as "one-level" dynamic branch prediction schemes. The BHT is a fully associative array of entries that stores a subset of the bits in each branch's address as a tag together with some history bits that predict the outcome of the corresponding branch(es). Ideally, the prediction made depends only on the history of a single branch but it is, of course, possible that more than one branch may map to the same entry in the

BHT. In this case, the history bits *may* reflect a mix of the branching behaviour of the branches mapped to that entry.

In two-level branch prediction schemes ([49][63]) the prediction is not only based on the past history of the branch under consideration but also the outcomes of other recently executed branches in the instruction stream. Pan, et al. [49] proposed a scheme that uses a correlation-based branch prediction approach implemented in hardware that associates multiple predictors with each conditional branch (or set of conditional branches) in a program. A shift register records the outcomes of the previously executed conditional branches, and this record is used to select one of the multiple predictors in a set for use in prediction. In this way, different control flow paths can lead to different branch predictions for the same branch instruction. This approach will serve as the basis for the proposed "path-based" object access predictor described later (see Section 4.5.4).

Hybrid Branch Prediction:

Young and Smith [65] proposed a compile-time, profile-based algorithm for predicting branches statically using correlation information gathered at runtime. Their profiler collects not only dynamic branch statistics (i.e. taken vs. not taken counts) for each branch in a program but also the taken vs. non-taken tendencies along each execution path that reaches the branch. This path information is essentially encoded in the program counter (via code duplication and modification) and is thereby used to implement the prediction when the program runs.

Tarlescu, et al [56] proposed an Elastic History Buffer scheme that can exploit the property that each branch instruction may have a different degrees of correlation with other branches, while maintaining the structure of a global branch history buffer. The number of bits from the global branch history buffer determines the degree of correlation when predictions are made. This number is encoded in the branch instruction and is derived from branch characteristics provided by a preliminary " profiling" run.

3. Problem Description

The latency of data access (object or otherwise) in distributed systems which ship data rather than operating on it remotely is a critical design issue. When implementing such a distributed system based on persistent objects, the objects must, of course, reside in a local machine's memory before they can be manipulated. To make objects memory-resident they first have to be fetched from a remote machine. This introduces even greater latency (sometimes by orders of magnitude) to object reference than are already seen in non-distributed systems. As a result, object replication, caching, and pre-fetching have all been proposed to minimize the effective network latency involved in remote object access.

As described earlier, several strategies have been previously proposed to pre-fetch objects, mostly in the context of object databases[5]. These techniques consider exploiting user-hints [12], retrieving previously clustered objects from a page server[1][23], and exploiting the semantic structure of objects [8]. Techniques based on the analysis of recent object access patterns, however, have received little or no attention in the literature. The research problem that this thesis seeks to explore is how to decrease the average object access latency in a distributed persistent object system using object pre-fetch strategies that collect, store, and eventually exploit information on recent inter-object access patterns.

3.1. The Assumed Environment

Much interest has been expressed recently in both persistence and distribution for object programming environments [59]. Adding persistence and transparent distribution to object systems offers significant benefits including a much-simplified programming environment, and improving sharing and potentially improved performance. In the early nineties, Graham, et al. [22] proposed a distributed shared virtual memory (DSVM)

[5] The pre-fetching is done from disk rather than across a network.

system, which supports persistence and distribution. Their goal is to build a distributed persistent object system in a 64-bit, shared address space. Objects are placed in the memory and thereby become transparently shared and accessible across multiple interconnected machines in a distributed environment. Thus, the persistent object space is visible to all processors at all sites and the distribution is invisible to users who access objects simply by invoking methods on them at their persistent locations in the DSVMS.

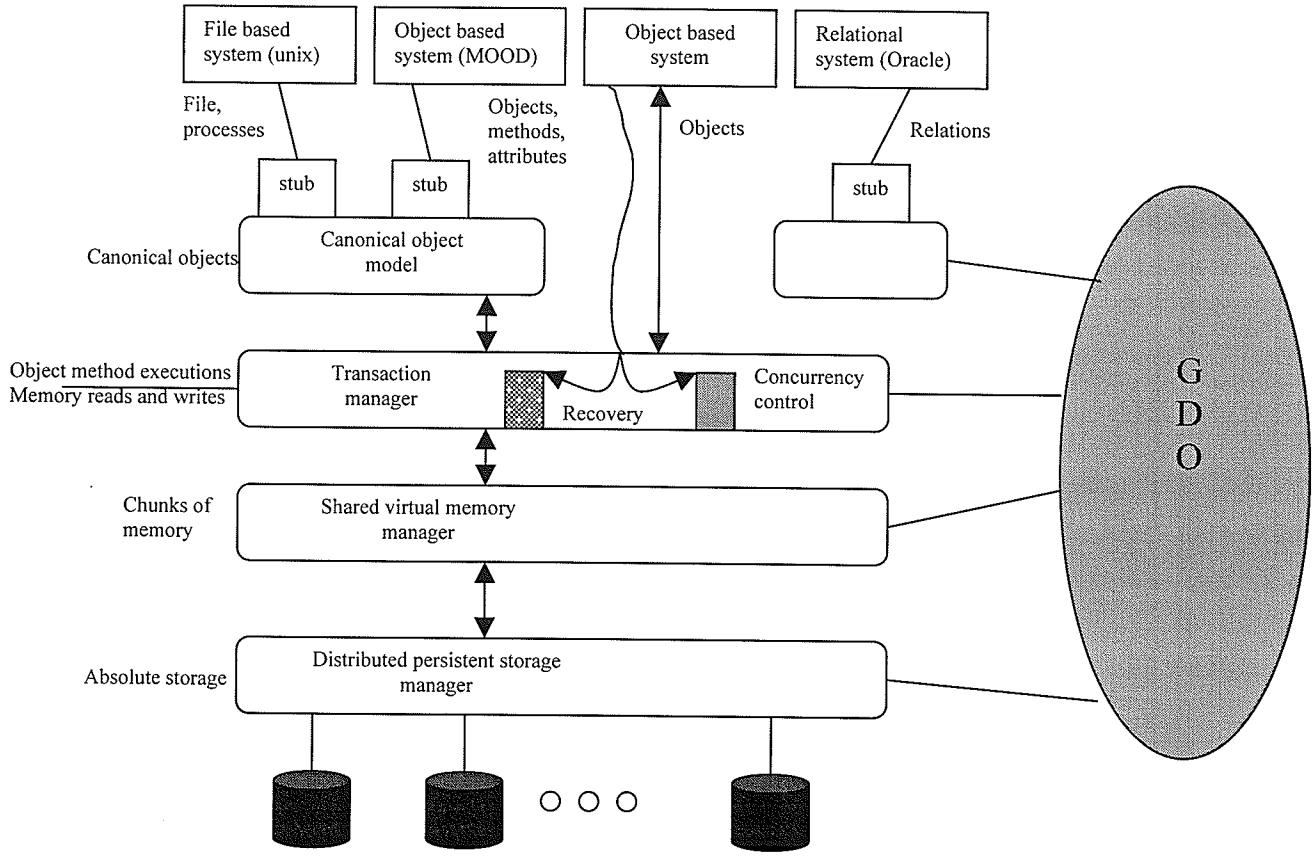

Figure 3.1 - System Architecture of a DSVM System

The system architecture proposed by Graham, et al. [22] is shown in Figure 3.1. As shown, a DSVM System includes a network, distributed persistent storage manager, persistent disk storage, shared virtual memory manager, transaction manager and a global directory of objects (GDO). A number of disks are distributed across several machines interconnected by a network and they collectively provide the system's non-volatile storage for the persistent storage of both objects and the GDO. A distributed persistent storage manager running over all disk sites manages these disks.

The GDO for the persistent object system is not only a repository of information about the objects being maintained in the system, but is also a provider of management services on those objects. The contents of the GDO in a DSVM system is described in [40]. The entries in the GDO must be distinguished from one another and explicitly associated with the objects they describe. This is done by using the OID of the associated object as a key in the GDO. In the case of a DSVM system, the OID for an object is simply the virtual address at which the object occurs in the persistent object space. Additional information included in the GDO may be divided into three basic categories: object representation information, replication sensitive object management information and replication insensitive object management information. For example, a GDO entry contains the addresses of object attributes and methods located on the nodes that persistently store the segments.

3.2. Predicting which Objects to Pre-fetch

Caching is an effective technique for decreasing data access latency. In the case of a DSVM system, each local machine's main memory is effectively used as an object "cache" (i.e. the main memory is caching objects from the global shared object space). Unfortunately, the cost of "cache misses" in such an environment is extremely high. As described earlier, the goal of data pre-fetching is to anticipate potential "cache misses" and fetch data before the processor needs it. Using pre-fetching in a DSVMS, the latency of object access can be decreased by ensuring that required objects will exist in the local memory "cache" before they are needed. Overall efficiency is improved since each processor in the system will be busy doing other computations while objects that will soon be required are being fetched from remote disks (i.e. object transfer occurs concurrently with computation).

To pre-fetch objects effectively, it must be possible to predict in advance which objects are likely to be needed in the near future. Good prediction, of course, should be able to significantly improve system performance since the high penalty of misses will be avoided. There is, however, a danger with object pre-fetching. Poor prediction may actually result in performance degradation due to additional network overhead and due to the loading of unnecessary objects into memory. Thus, caution must be used in designing any object pre-fetching strategy. Some of the issues that need to be considered include:

- Accuracy of prediction
- Timing of pre-fetching
- Flexibility of prediction policy
- Cost of prediction

Prediction of data/object access patterns could be based on many things including: spatial locality, temporal locality, historical object access patterns, and object semantics. No matter which strategy is employed, the core issue is the accuracy of prediction. Incorrect predictions could result in a large number of unneeded objects being loaded into a machine's memory from remote disks in a distributed OO system. This would lead to performance degradation due to the time wasted for the transmission of objects through the network (which also affects other, unrelated, computations in the system). Further, some resident objects that might be useful in the future may have to be evicted to make room for the pre-fetched object, which, in the worst case due to inaccurate prediction, might never be used. In this case, the cache is said to have been "polluted" with non-useful objects.

The timing of pre-fetching also affects the overall performance of the system. Ideally, prediction and pre-fetching will be completed while the processor is doing other computations so that the next object to be referenced will be fetched into memory just before the processor needs it. If such objects arrive too early, cache pollution occurs as some potentially useful objects may be evicted to accommodate the new pre-fetched objects. On the other hand, if the pre-fetched objects arrive too late, the objects won't be in the cache when the processor needs them and at least some portion of the object access latency will not have been hidden.

Flexibility and the cost of the prediction policy should also be considered when designing a pre-fetch scheme. In a distributed OO system, new objects could be added with the growth of a company, or to reflect updates to an existing system. This requires that the prediction policy be flexible enough to accurately predict future object accesses even in an environment where the set of objects and their interactions change dynamically. Further, object pre-fetching, when done carefully, certainly has the potential

to improve overall system performance but it does not come for free. There are costs associated with performing both the prediction and pre-fetching. These costs should be minimized whenever possible and certainly should be constrained to being a small fraction of the benefit offered by the pre-fetching.

3.3. Software Support for Object Pre-fetching

Despite the advantages provided by such distributed persistent object systems, widespread adoption has been slow partly because efficient implementation remains an issue [15]. Intelligent object pre-fetching is a technique that may be used to address this problem. The question is how best to facilitate such "intelligence" in the pre-fetching process. In a page-based object system, object pre-fetching is effectively achieved by retrieving all the objects residing in the same page as an object of interest. The accuracy of object pre-fetching using this strategy is entirely dependent on the extent to which objects are correctly clustered into pages. Pre-fetching by using advanced object semantics such as inheritance and other structural relationships offers potentially improved performance, but few studies have been reported that investigate the necessary system software support for such pre-fetching. This support includes the data structures and algorithms used to collect and store information on recent access pattern as well as techniques to use the information to do the actual pre-fetching. These are the specific subjects of this thesis.

4. Problem Solution

In this section, six specific strategies for object pre-fetching are proposed. All the strategies are variations on three basic concepts presented below. Each of these concepts is based on exploiting historical information about inter-object method invocation patterns (in the distributed persistent object space). A "reference predictor" that encapsulates dynamically updated information on recent method invocation patterns and stores it in the entries of the GDO (Global Directory of Objects) is common to all the strategies. What differs is the access pattern information collected and how it is eventually used. The goal of the reference predictor is, of course, to improve overall system performance by predicting objects that are likely to be needed in the near future so that they may be pre-fetched from remote locations thereby hiding the network latency of the necessary transfer. The policies, data structures and algorithms proposed for object pre-fetching are described in the following sections. The techniques proposed extend some of the work related to object and data pre-fetching and branch prediction, presented earlier in Section 2.2 by applying the idea of tracking historical access patterns to the method invocation relationship. Specifically, the concepts of "pre-fetching threshold", "pre-fetching depth" and "path-based pre-fetching" are defined and used in the object prediction process to produce a family of pre-fetch strategies offering tradeoffs between cost, accuracy and efficiency in object prediction.

4.1. Policies for Object Prediction

Although the pre-fetching strategies proposed here were designed to be implemented in a distributed persistent object-based system implemented in a shared virtual address space (i.e. DSVMS), the ideas and data structures used for object prediction can also be applied to other systems with mobile objects[6]. The object model assumed in this thesis is simple. Each persistent object has a unique Object IDentifier (OID), a structural component consisting of its attributes (data), and a behavioral

[6] Mobile objects are those which can move from machine to machine in order to allow method invocations to execute locally.

component consisting of a set of methods that manipulate the attributes of the object. An object invokes an operation in another object by communicating with it through a method invocation. These basic features are common to all object-oriented programming environments and thus the results presented here should be generally applicable to all persistent distributed object systems.

When devising strategies for object pre-fetching, decisions have to be made concerning:

i) Which object relationship(s) will be used for the prediction,

ii) Which objects will be pre-fetched, and

iii) Where and how to collect the information needed for object prediction.

To discuss pre-fetching conveniently in the rest of the thesis, some basic notation is now introduced. In the rest of this thesis:

O will denote any object in the persistent object system.

{O} will denote a set of objects in the persistent object system.

Oi or **{Oi}** will, by convention, generally denote the object or set of objects being accessed by the currently executing process.

Oj or **{Oj}** will, by convention, generally denote the next object or next set of objects that are likely to be accessed by the currently executing process.

m will denote a method on an object.

{m} will denote a set of methods on an object.

Further, the following notation is used to indicate that the invocation of method, m, on object, Oi, should lead to a pre-fetch of Oj into the local memory for execution.

$$Oi \xrightarrow{m} Oj$$

4.2. A Generic Reference Predictor

The data structure used for data or object access prediction has been called by a variety of names including "history table", "reference table", "prediction table", or just

"predictor" (depending on the area of application). In this thesis, object access prediction is done using a per-object "reference predictor" that is associated with each object's GDO entry[7]. The basis of all the pre-fetching algorithms presented in this thesis are algorithms to determine the "next" object(s) that might be referenced and to dynamically track which object(s) actually are referenced over a period of time. For each object/method, a "reference weight" is associated with each likely next object to be referenced that reflects the probability that it will be referenced (based on past reference behavior). These reference weights are then used in predicting which objects to pre-fetch.

Each reference predictor, therefore, consists of a data structure enumerating possible "next-accessed" objects and their associated weights. When an object Oi is accessed, the information stored in the reference predictor will be examined and the object(s) with higher weights (which are predicted, with high probability, to be accessed next) will be pre-fetched into the local machine's memory. After some next object is accessed, the system must increase the weight of the corresponding object in the reference predictor in preparation for the next pre-fetch. Figure 4.1 demonstrates this simple form of object prediction. In the figure, Oi is the object upon which a method is being executed; entries for the objects Od, Oj and Ok reside in the reference predictor since these objects were referenced after Oi in the (recent) past, and the numbers in the small circles represent the weights associated with each corresponding object[8]. The object with the highest weight (Oj in the example) is the object that is selected for pre-fetching.

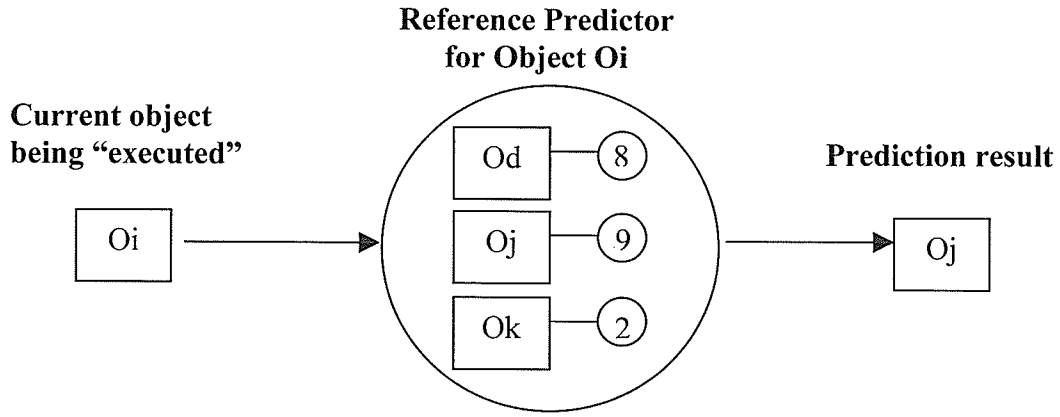

Figure 4.1 - Object prediction using reference weights

[7] By making the reference predictor a part of the GDO the issue of distributing the predictor to each machine in the distributed system can be ignored since this is an existing GDO service.
[8] Again, the weights reflect the probability that the corresponding object will be accessed next.

4.3. Incorporating Reference Predictors into the GDO

The GDO in a DSVM system acts as a repository of information about the objects being maintained in the system. It also serves as a provider of management services on those same objects. Finally, it also provides for the automated distribution of copies of the data that it stores which is safely replicable to local machines to enhance execution efficiency and coordinates the update of GDO data to ensure consistency despite the distributed nature of the system.

The entries in the GDO must be distinguished from one another and explicitly associated with the objects they describe. This is done using the virtual address at which each object occurs in the persistent object space (i.e. each object's OID). As described by Mathew, et al. [40], a GDO entry contains the on-disk addresses of the corresponding object's attributes and methods as well as information concerning object-class relationships, etc. The reference predictor for each object will also be incorporated into the corresponding GDO entry. Then, when a transaction invokes a method on some object, Oi, the GDO entry will be looked up using the current object's OID as the key. The reference predictor for that object will then be made available to the machine operating on the object so that the necessary prediction computation and eventual pre-fetching may be done.

4.4. Pre-fetching Threshold, Depth and Path-based Pre-fetching

The basic pre-fetch technique described in section 4.3 can (and should) be extended in several ways to improve pre-fetching. The crucial issues for object pre-fetching are the accuracy of the prediction and the timing of bringing required objects into the memory. Incorrect predictions could result in a large number of unneeded objects being brought into the machine's memory from remote machines. As described earlier, this leads to performance degradation due to both wasted time and bandwidth in transmitting the objects over the network and due to cache pollution, which occurs because useful objects may have to be evicted from the memory to make room for incoming objects. To improve the accuracy and efficiency of object prediction the following concepts are proposed (and later incorporated into the six pre-fetching strategies described in this thesis):

i) Pre-fetching threshold,

ii) Pre-fetching depth, and

iii) Path-based pre-fetching.

Pre-fetching threshold:

In the reference predictor, the probability that an object will be referenced is determined by its associated stored weight. The most likely candidate for pre-fetching, therefore, has the highest weight whereas the least likely candidate has the lowest weight. In the example in Figure 4.1 object Oj alone was selected because it had the highest weight. Note, however, that object Od had a weight that was nearly as high as Oj's. Perhaps it would be advantageous to pre-fetch both Oj and Od. This raises the question of how exactly the stored reference weights should be used to select objects for pre-fetching.

The pre-fetching threshold is defined as the minimal weight that must be associated with an object for it to be pre-fetched. In practice, the reference predictor should initiate a pre-fetch for an object only if its weight is above a given pre-fetching threshold, thereby keeping the number of pre-fetched objects relatively low (to avoid cache pollution) while hopefully still being more accurate than is possible pre-fetching a single object alone. The pre-fetching threshold need not be an absolute value. It can be determined in a number of ways considering such factors as relative weights (e.g. "fetch those object with weights that are significantly higher than others") and number of objects (e.g. "fetch no more than three objects"), etc. as well as combinations of these concepts. Generally, a pre-fetching threshold will be selected using some sort of cost/benefit parameter such as those discussed in [31].

Pre-fetching depth:

It may be that the pre-fetching of just the "next" object(s) to be accessed will provide only limited benefit. This would be true, for example, if the pre-fetched object performs minimal processing and then quickly invokes methods on other objects. Consider an object Oi, which is executing and for which Oj is predicted. Assume also that Ok is predicted for Oj. Because the pre-fetched object (Oj) executes only briefly, there is little time to pre-fetch the object(s) that are predicted for it (Ok in the example). Thus, the second "stage" of pre-fetching is likely to be ineffective since there will be

insufficient time to bring Ok into the local machine's memory (i.e. the pre-fetch will be too late) because the execution time for Oj is smaller than the reference latency for Ok.

To improve system performance in such situations, the concept of pre-fetching "depth" can be used. Instead of having each reference predictor store a set of single objects, they can store a set of sequences of objects. In the previous example, we might store a sequence (of length 2) in the predictor for Oi, which would consist of "Oj →Ok" indicating that after Oi, Oj was likely to be accessed and after Oj, Ok was likely to be accessed. Given such information, a pre-fetch depth of either 1 or 2 could be specified. If a depth of 2 was specified, then both Oj and Ok would be pre-fetched for Oi. If a depth of 1 was specified, only Oj would be pre-fetched.

Thus, for each object or method invocation[9], recent sequences of object accesses should be collected and sorted according to their "closeness" to the current object. For example, the set of objects which may be directly invoked by the current object, Oi, are defined to have prediction depth 1 (relative to Oi), the set of objects which may be directly invoked by the objects at depth 1 are defined to have prediction depth 2 relative to Oi, etc. Within each depth level, the objects with weights greater than some threshold will be selected as candidates for pre-fetching. Assuming the pre-fetching depth is set to 3, when Oi is accessed, candidates from depth 1, depth 2 and depth 3 will all be pre-fetched. The extent of pre-fetching can thus be determined by the pre-fetching depth and pre-fetching threshold as needed. Numerous options exist for how to combine pre-fetching thresholds and pre-fetching depth. In particular, it may be desirable to link the pre-fetching of objects from depth 'i' to the earlier pre-fetching of related objects (i.e. in depths 'i-1', 'i-2', etc.). This is since pre-fetching an object at depth 'i' which will be referenced only if some other object is pre-fetched earlier only makes sense if the earlier pre-fetch is actually done. Figure 4.2 demonstrates a simple use of the concept of pre-fetching depth where only the most likely object at each depth is pre-fetched.

[9] Pre-fetch information can be stored per-object or per-object *method*. The latter should provide more accurate pre-fetching as methods may reference different objects but incurs higher predictor storage cost.

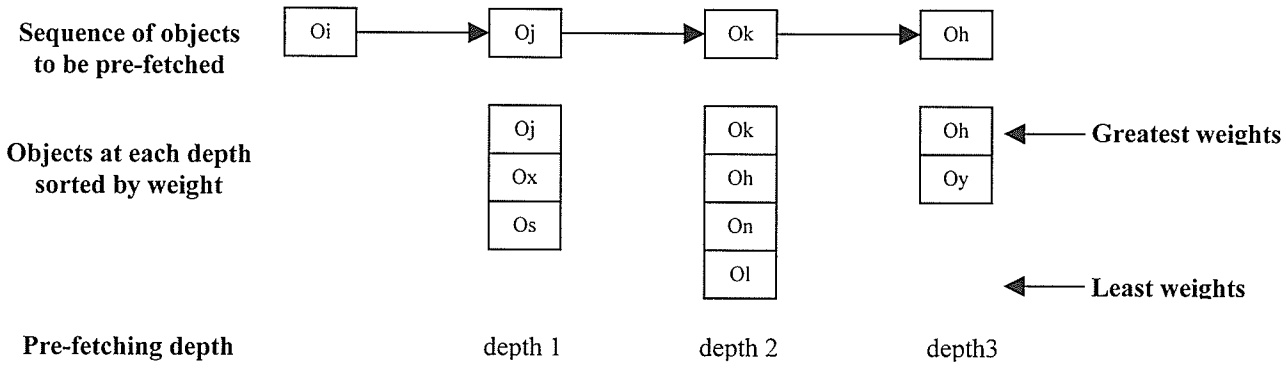

Figure 4.2 - Application of pre-fetching depth

Path-based Pre-fetching:

Just as object access patterns may change from method to method so too may they change depending on how a single method is invoked. This is because different invocation "paths" may lead to different changes in input conditions (e.g. method arguments), which can affect a method's object referencing behaviour. This is similar to the concept used in two-level branch prediction discussed in Section 2.2.4. Recognizing that method invocation behaviour may change based on execution context provides another opportunity for improving the effectiveness of object pre-fetching.

The concept of "path-based" pre-fetching is an extension of previous work on branch-directed data pre-fetching (e.g. [39]) into persistent object systems. Path-based pre-fetching allows object pre-fetching to vary based on the object invocation path that lead to the current object's method execution. In a persistent object system a "path" is simply defined to be a sequence of method invocations made as part of a program's execution. For example, a given method invocation, m, on object Oi may have arisen due to an Object Oj invoking that method or due to another object Ok invoking that method. Of course, this generalizes to a sequence of objects invoking methods on one another that eventually results in method m being invoked on object Oi.

To support path-based pre-fetching, historical path information must be recorded in each reference predictor. Then, when an object is executed, the reference predictor will search through the available path histories to find a match with the current execution path and select the appropriate pre-fetch information to predict which objects will

subsequently be accessed. Path-based pre-fetching may also be combined in various ways with the concepts of pre-fetching threshold and pre-fetching depth.

4.5. Strategies for Object Prediction

Clearly, there are many different ways in which pre-fetching threshold, pre-fetching depth and path-based pre-fetching may be combined and, further, each of these concepts may themselves be parameterized (e.g. different threshold values). Thus, these concepts introduce a large family of potential pre-fetching strategies. It is not possible, nor desirable, to enumerate "all" such strategies in this thesis. Thus, in this section, six specific pre-fetch strategies have been selected which are described in detail including the data structures and algorithms used for tracking object behavior and making pre-fetch predictions. The strategies are described in order from the simplest to the most complicated.

For each of the six strategies, two algorithms must be described: one that updates the data structure(s) in the reference predictor so that they store accurate access information, and another that describes how to use the stored information to actually perform the pre-fetching. Additionally, declarations of the data structures themselves must be provided. A C-language like pseudo code is used in presenting the algorithms for each of the strategies. Three figures will be provided for each strategy providing the data structure declarations, data structure update algorithm and the pre-fetching algorithm in that order.

4.5.1. Strategy 1: $O_i \rightarrow O_j$

Strategy 1 predicts a single object that is likely to be accessed "next" given the current object. It is the simplest and, in most cases, likely the least-effective strategy.

Data structure for strategy 1:

Strategy 1 corresponds to the "generic predictor" illustrated in Figure 4.1 In this case the reference predictor must store a collection of possible "next" objects and their associated probabilities. Each reference predictor thus consists of an ordered array of two-element structures corresponding to the objects that may be accessed next. The first element of each such structure stores the OID of the object likely to be pre-fetched and

the second element stores the corresponding weight associated with that object. An example of the data structure used in the reference predictor is shown in Figure 4.3.

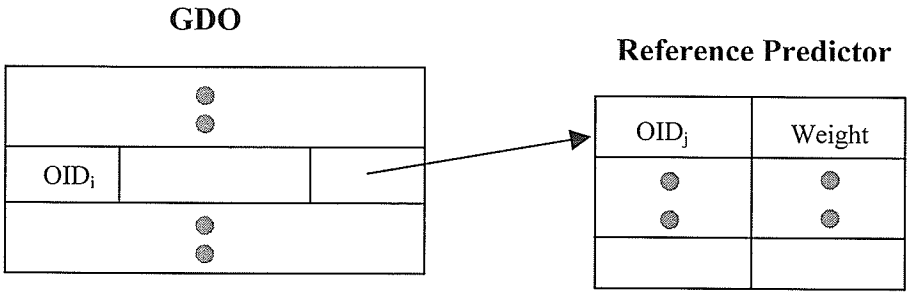

Figure 4.3 - Data structure for Strategy 1

Object prediction for strategy 1:

When the current object (Oi) is accessed only a single potential next object (Oj), the one with the highest weight based on the history of recent accesses, will be pre-fetched. When Oi is accessed, the entry with OIDi will be looked up in the GDO. The associated reference predictor will be consulted to determine which object has the highest weight and that object will be pre-fetched into the local machine's memory.

In addition to supporting pre-fetch prediction, the reference predictor data structure must also support updating the weights associated with the potential next objects based on which object is accessed. This is done based on the outcome of the current object execution. Once the actual next object to be executed is determined, the corresponding weights in the reference predictor must updated as follows:

1. If the entry's OIDj field matches the OID of the actual next object referenced, then the corresponding weight field is increased by 1 up to a fixed maximum[10].
2. If the entry's OIDj field does not match the OID of the actual next object decrease the corresponding weight field in the array by 1 down to a fixed minimum (presumably 1).
3. If there is no entry for the actual next object referenced in the predictor and the array containing the pre-fetch candidates isn't fully filled, add the OID of

[10] The maximum will be determined to avoid exceeding the representation ability of the weight field.

the actual next object referenced into the first vacancy in the array and set the corresponding weight field to the average of the existing weights in the array. This permits new entries to be dynamically added to the reference predictor.

This strategy is easy to implement and is also highly efficient (i.e. low runtime overhead) and is accurate when there is a very consistent and specific relationship between the current and next objects referenced. However, in more complicated (and realistic) object systems, the access patterns between objects will likely be variable and therefore pre-fetching only the one candidate object with the highest weight could lead to high miss rates and corresponding performance penalty.

Algorithms for strategy 1:

Before discussing the algorithms for strategy 1, the data structure used to store the historical reference information must be described.

```
#define MAXentries "max number of next objects"

typedef struct rpentry {
   OID Oid=-1;                  // initially no OID
   int weight=0;                // initially no weight
} *RefPredictorEntry;

typedef struct pred {
   RefPredictorEntry entries[MAXentries];
} *Predictor;
```

Figure 4.4 - The Predictor Data Structure for Strategy 1

As shown in Figure 4.4, each GDO entry will contain an entry of type Predictor which is a reference to the predictor for the corresponding object. Each Predictor, as described earlier, consists of an array of entries each of which is a <nextOID, weight> pair. For convenience, the array is defined to have a fixed maximum size (MAXentries) but this could, of course, equally well be implemented as a dynamic array (e.g. the Java Vector class) to accommodate arbitrarily few or many entries. Note that each weight is initialized to zero. This is because the weights will be set dynamically over time in response to actual object reference patterns as outlined previously. Further, it does not make sense to assign an actual weight to an entry for which the object identifier is not yet set (since OIDs are also set dynamically).

```
Predictor pred=GetPredictor(CurrOID);
OID nxt = object identifier of next accessed object;

// look for next object's OID and inc. weight if found
int found=0;
for (int i=0; i<MAXentries; i++) {
   RefPredictorEntry p=pred.entries[i];
   if (p->Oid==nxt){
        p->weight++; // increase weight on hit
        found=1;
   } else if (p->Oid!=-1){
        p->weight--; // decrease weight on miss
        if (p->weight==0){
             // mark slot as free
             p->Oid=-1;
        }
   }
}

if (found==0){ // next referenced object not found
// add next object's OID to array if space is left
   if (pred.entries[MAXentries-1]->Oid==-1){
        // array is not full
        int Avg=0;
        for (i=0; i<MAXentries; i++) {
             RefPredictorEntry p=pred.entries[i];
             if ( p->Oid== -1){  // found space
                  if (i!=0) {
                       Avg=Avg/i;
                       p->Oid=nxt;
                       p->weight=Avg;
                  } else {
                       p->Oid=nxt;
                       p->weight=MAXINT/2;
                  }
                  break;
             } else {
                  Avg=Avg+p->weight;
             }
        }
   }
}
```

Figure 4.5 - Updating the Predictor Data Structure for Strategy 1

The code in Figure 4.5 illustrates the processing needed to maintain the weights associated with the objects that Oi has recently invoked methods on. Initially the OIDs of all entries in the array maintained by the reference predictor are set to –1. The code shown then supports the dynamic addition of new entries in the reference predictor by locating an empty entry (which has an OID of –1). The code also increments the weight

for the next-referenced object and (in the same loop) decreases the weights associated with the non-referenced objects. In this way, the *relative* weights of objects are maintained in a way that is easy to implement "on the fly". Simply incrementing and decrementing the weights is not the only option. It is certainly possible to change the amounts by which the weights are incremented and decremented in order to ensure that the weights accurately reflect the probability of each object being accessed next. For example, with the aid of analysis of empirical data from an actual implementation, it might turn out to be useful to change the algorithm to add 10 on a hit and subtract 2 on a miss. The exact values that will best suit a given system can only be determined by direct observation of the system.

Note that while it would seem logical to set the weight for a newly inserted object to one (accurately reflecting how often it has been referenced next) this is not done. Instead, the weight for a newly inserted OID is set to be either the average weight of the entries already stored in the array or MAXINT/2 if this is the first entry being inserted in the array. This is necessary to ensure that new entries in the array are not immediately removed before they may be used. If a new object were to be added to the array with a weight of one, then unless it were to be accessed again, immediately, it would be immediately removed from the array (when its weight field is decremented to zero on the next object access). A pathological case illustrating this is where some object, O_i, alternates between accessing O_j and O_k next. In this case, neither O_j nor O_k would ever be pre-fetched because neither gets an entry in the array long enough to be used. Other problems can also arise if the weight of new objects is set to 1. For example, consider a 3-element array containing entries for the three objects O_a, O_b and O_c with corresponding weights of 3, 2 and 1, respectively. If the sequence of objects to be accessed next is O_d, O_e and O_f then if the weights are decreased by one every time there is a miss then after the reference to object O_f it will be impossible to tell that object O_a should be preferred for pre-fetching over object O_f (since they will both have stored weights of one). The setting of the initial object weight as described above ensures that object access information is not discarded too soon.

```
Predictor pred=GetPredictor(CurrOID);
RefPredictorEntry q=pred.entries[0];
OID PrefetchOid = q->Oid;
int BiggestWt = q->weight;

for (int i=1; i<MAXentries; i++) {
   // search array to find element with biggest weight
   RefPredictorEntry p = pred.entries[i];
   if (p->weight > BiggestWt){
        PrefetchOid = p->Oid;
        BiggestWt = p->weight;
   }
}
if (PrefetchOid!=-1){
   Pre-Fetch(PrefetchOid);  // do the pre-fetch
}
```

Figure 4.6 - Using the Predictor for Strategy 1

Using the information in the reference predictor is straightforward and is illustrated in Figure 4.6. Since the entries in the reference predictor are not maintained in sorted order by the code that updates the reference predictor data structure, the entries in the predictor array must be searched to find the entry with the greatest weight.

4.5.2. Strategy 2: $o_i \rightarrow \{ o_j \}$

Strategy 2 is an extension of strategy 1, which pre-fetches a *set* of objects that are likely to be referenced "next" instead of pre-fetching a single object. While it is easy to select and pre-fetch the single next object with the highest associated weight (as is done in strategy 1) such a strategy will likely produce only limited benefit in complex object systems where object access patterns may not be consistent over time. In particular, if there are additional objects stored in the reference predictor that have a weight near to that of the object with the highest weight, then those objects will not be pre-fetched and performance may suffer as a result. To resolve this problem, strategy 2 predicts more than one object to be pre-fetched from the stored list of candidate next objects. To limit the number of objects that are actually brought into the local machine's memory, the concept of pre-fetch threshold (based on a weight threshold value) is applied in strategy 2. This will prevent the pre-fetching of those objects that might be accessed next but only with low probability. A weight-based threshold makes more sense than simply pre-fetching a pre-determined number of objects. Some objects may reference only a single "next" object with high probability while others may reference several. This means that

deciding to pre-fetch exactly k objects may be inadequate in some cases and be "overkill" in others.

Data Structure for Strategy 2:

The data structure for strategy 2 is the same as the one used for strategy 1 except that in strategy 2 the reference predictor collects the history information on possible "next" objects and sorts these objects in order of decreasing weight rather than simply storing them unordered as in strategy 1. Accordingly, the reference predictor stores a collection of possible "next" objects and their associated reference probabilities. Each reference predictor will thus consist of an *ordered* array of two-element structures corresponding to the objects that may be accessed next. The first element of each such structure will again store the OID of some object that is likely to be referenced next and the second element will store the corresponding weight (reflecting the object's probability of being accessed next) associated with that object. The objects in each reference predictor will now be sorted in order of decreasing weight so that it will be easy and efficient to select all objects with a weight exceeding a certain threshold value. An example of the data structure used in for strategy 2 is shown in Figure 4.7.

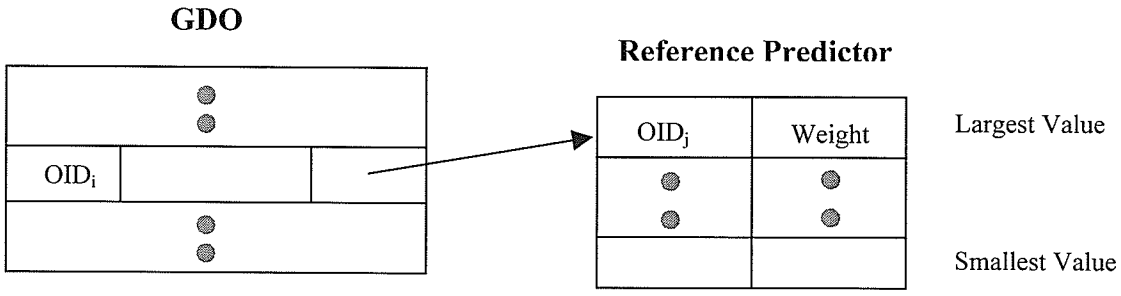

Figure 4.7 - Data Structure for Strategy 2

Object prediction for strategy 2:

When the current object, Oi, is accessed, a set of potential next objects {Oj} (determined by the object's recent access history) must be pre-fetched. To accomplish this, when Oi is accessed, the entry for OID i will be looked up in the GDO and the

associated reference predictor will be used to determine which set of objects should be pre-fetched into the local machine's memory.

In general, since |{Oj}| may be large, there is the possibility of bringing a large number of potential "next" objects into memory. Unless all such objects will be referenced next, this will result in cache pollution because some objects that may be useful in the future will likely have to be evicted to provide space for the newly pre-fetched objects. This raises the question of how exactly to use the stored information in the reference predictor to prune the number of objects that will be brought into the local machine's memory (i.e. how to implement a pre-fetch threshold).

The concept of pre-fetching threshold is implemented in strategy 2 using a percentage threshold value. The object with the highest weight is always pre-fetched and other objects stored in the reference predictor structure will be pre-fetched only if their weights are within a given percentage of the highest weight. Using pre-fetch threshold in this way will keep the number of predicted objects (and therefore the cache pollution) relatively low while also minimizing the miss rate (by always selecting only useful objects for pre-fetching).

In addition to supporting pre-fetch prediction, the reference predictor data structure must also support updating the weights associated with the potential next objects based on which object is accessed. This is done based on the outcome of the current object's execution. Once the actual next object to be executed is determined, the corresponding weights in the reference predictor must be updated as described in Strategy1.

Overall, strategy 2 should be both efficient and more accurate in its pre-fetching behaviour for more complicated object systems where pre-fetching a single object is insufficient.

Algorithms for strategy 2:

The reference predictor data structure and the algorithms (for updating and using the reference predictor) for strategy 2 are now presented.

```
#define MAXentries "max number of next objects"

typedef struct rpentry {
   OID Oid=-1;                  // initially no OID
   int weight=0;                // initially no weight
} *RefPredictorEntry;

typedef struct pred {
   RefPredictorEntry entries[MAXentries];
   float ThresholdPctg;     // value between 0 and 1
} *Predictor;
```

Figure 4.8 - The Predictor Data Structure for Strategy 2

As shown in Figure 4.8 the data structure for strategy 2 is the same as that for strategy 1, except that a new field `ThresholdPctg` in the `Predictor` structure has been added which stores a threshold percentage value (in the range 0 to 1). For an object to be pre-fetched, its weight must be within `ThresholdPctg` percent of the highest weight value. Using this approach should, as described previously, result in more effective pre-fetching than using a threshold value that directly determines the *number* of objects to be pre-fetched.

As in strategy 1, each GDO entry will also contain an entry of type `Predictor`, which is a reference to the predictor for the corresponding object. Each `Predictor`, as described earlier, consists of an array of entries each of which is a <nextOID, weight> pair. For convenience, the array is defined to have a fixed maximum size (`MAXentries`). Each weight is, again, initialized to 0 and each OID is initialized to -1.

```
Predictor pred=GetPredictor(CurrOID);
OID nxt = OID of the next object actually accessed;

// look for next object's OID and inc. weight if found
int found=0;
for (int i=0; i<MAXentries; i++) {
   RefPredictorEntry p=pred.entries[i];
   if (p->Oid==nxt){
        p->weight++;
        // the increment may have changed the order
        // of elements in the sorted array so...
        for (int j=i; j>0; j--) {
             //make the array ordered by the weights
             RefPredictorEntry x=pred.entries[j-1];
             RefPredictorEntry y=pred.entries[j];
             RefPredictorEntry tmp;
             if (x->weight<y->weight){ // swap
                  tmp->Oid=y->Oid;
```

```
                    tmp->weight=y->weight;
                    y->Oid=x->Oid;
                    y->weight=x->weight;
                    x->Oid=tmp->Oid;
                    x->weight=tmp->weight;
               } else{
                    break;
               }
          }
          found=1;
     } else if (p->Oid!=-1){
          p->weight--; // decrease weight on miss
          if (p->weight==0){
               p->Oid=-1;   // mark slot as free
          }

     }
}

if (found==0){ // next referenced object not found
  // add next object's OID to array if space is left
   if (pred.entries[MAXentries-1]->Oid==-1){
          // array is not full
          int Avg=0;
          for (i=0; i<MAXentries; i++) {
               RefPredictorEntry p=pred.entries[i];
               if ( p->Oid== -1){  // found space
                    if (i!=0) {
                         Avg=Avg/i;
                         p->Oid=nxt;
                         p->weight=Avg;
                    } else {
                         p->Oid=nxt;
                         p->weight=MAXINT/2;
                    }
                    for (int j=i; j>0; j--) {
                         // reorder for new entry
                         RefPredictorEntry x, y, tmp;
                         x=pred.entries[j-1];
                         y=pred.entries[j];
                         if (x->weight<y->weight){
                              tmp->Oid=y->Oid;
                              tmp->weight=y->weight;
                              y->Oid=x->Oid;
                              y->weight=x->weight;
                              x->Oid=tmp->Oid;
                              x->weight=tmp->weight;
                         } else{
                              break;
                         }
                    }
                    break;
               } else {
                    Avg=Avg+p->weight;
```

```
                }
            }
        }
    }
```

Figure 4.9 - Updating the Predictor Data Structure for Strategy 2

The code in Figure 4.9 illustrates the processing needed to maintain the weights associated with the objects that Oi has recently invoked methods on for strategy 2. Initially the OIDs of all entries in the array maintained by the reference predictor are set to –1. The code shown again supports the dynamic addition of new entries in the reference predictor. As in strategy 1, the code must increment the weight of the next-referenced object and decrease the weights associated with the non-referenced objects, however, the fact that the array is now sorted by weight adds some complexity. When the OID of the next accessed object is found in the predictor array, the associated weight of that OID is increment by 1 and then the code ensures that the predictor array is still ordered by, if necessary, moving the entry for the next accessed object upward in the array to its appropriate position according to its new weight. As in strategy 1, if the OID of the next accessed object is not found in the predictor array (`'found==0'`), then an entry for the OID is created in the first available entry in the array (if there is one).

```
Predictor pred=GetPredictor(CurrOID);
RefPredictorEntry p;
int BestWeight;

// always pre-fetch the first object if it exists
p = pred.entries[0];
if (p->Oid!=-1) {
   Pre-Fetch(p->Oid); // do the pre-fetch
   BestWeight=p->weight;
} else {
   exit; // no pre-fetching to be done yet
}

// pre-fetch other "next" objects which have
// weights within ThresholdPctg of the first
for (int i=1; i<MAXentries; i++) {
   p = pred.entries[i];
   if (p->Oid==-1)
        break;
   if (p->weight >= (pred->ThresholdPctg*BestWeight)){
        Pre-Fetch(p->Oid); // do the pre-fetch
   else
        break;
}
```

Figure 4.10 - Using the Predictor for Strategy 2

Using the information in the reference predictor is relatively straightforward and is illustrated in Figure 4.10. Recall that the entries in the reference predictor are maintained in descending order (by weight) by the code that updates the reference predictor data structure. We always pre-fetch the first entry in the array (if it exists) and then pre-fetch all other entries in the array that have weights that are "close" to the weight of the first entry, where closeness is judged using the `ThresholdPctg` field.

4.5.3. Strategy 3: $O_i.M_k \rightarrow \{O_j\}$

Strategy 3 predicts a set of objects that are likely to be accessed "next" given the current object *method.* As described earlier, methods specify the behavioral component for each object. Each method performs its own, independent computations (including invocations of methods on other objects) and thus, an object's access patterns clearly may change depending on which method it is executing. The predictions made in strategy 1 and strategy 2 are based on recent object access patterns without knowledge of which object method made the invocations. Thus, for example, if some method m1 on object Oi is frequently invoked and it invokes on or more methods on object Oj then object Oj will be predicted for pre-fetching whenever a method is executing on Oi. If the executing method happens to be m1, this is good. If it happens to be some other method, say m2, which does not access Oj then it is bad since Oj will probably be pre-fetched unnecessarily. Clearly, considering which method(s) reference which other object(s) is important to achieve accuracy in such situations. To improve prediction accuracy, strategy 3 considers which method is executing when predicting which objects are likely to be accessed next. This will provide more accurate pre-fetching when the sets of objects referenced by different methods are themselves different at the expense of having to track next-access information on a per-object-method basis rather than just a per-object basis.

Data Structure for Strategy 3:

In strategy 3, the reference predictor for an object, Oi, stores a collection of possible "next" objects and their associated probabilities for each of the methods of object Oi. This is because the invocation of each method on an object can, of course, lead to separate sets of objects to be pre-fetched. Each reference predictor thus consists of an array representing the methods of the corresponding object. Each entry in that array refers

to an ordered array of two-element structures storing the information about the objects that may be accessed next. The first element of each such structure stores the OID of the object likely to be pre-fetched and the second element stores the corresponding weight associated with that object (as in strategies 1 and 2). An example of the data structure used in the reference predictor for strategy 3 is shown in Figure 4.11.

Object Prediction for Strategy 3:

Strategy 3 uses knowledge of the method, which is executing on the current object in the pre-fetching process. Thus, when method k of object Oi is invoked, the OIDi will be used to look up the appropriate entry in the GDO. The associated reference predictor will then be consulted to determine which set of objects is associated with method k of object Oi. A subset of those objects will then be pre-fetched into the local machine's memory considering pre-fetch threshold using the same basic technique as in strategy 2.

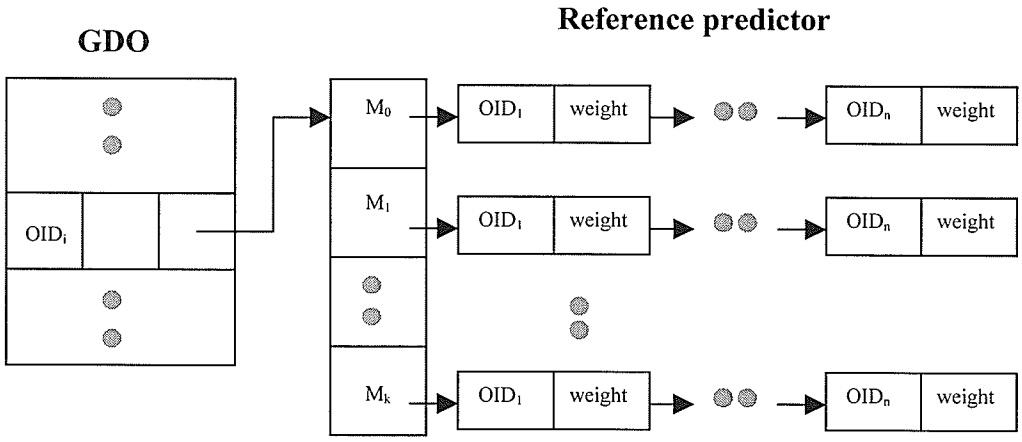

Figure 4.11 - Data structure for Strategy 3

By predicting next objects in a method-specific fashion, strategy 3 will allow for more accuracy and efficiency in pre-fetching in situations where the set of next objects that are likely to be referenced varies depending on which method is being executed. The additional cost of doing this, relative to strategy 2 is quite low. As can be seen in Figure 4.11 all that is required during pre-fetching is a single additional de-reference to find the relevant collection of <nextOID, weight> pairs. Of course, each reference predictor will have to store significantly more information.

Algorithms for Strategy 3:

```
#define MAXentries "max number of next objects"
#define MAXmethods "max number of methods / object"

typedef struct rpentry {
   OID Oid=-1;     // initially no OID
   int weight=0;  // initially no weight
} *RefPredictorEntry;

typedef struct OneMeth{
   RefPredictorEntry entries[MAXentries];
} *OneMethod;

typedef struct pred {
   OneMethod meth[MAXmethods];
   float ThresholdPctg;     // value between 0 and 1
} *Predictor;
```

Figure 4.12 - The Predictor Data Structure for Strategy 3

As shown in Figure 4.12, each GDO entry will contain an entry of type `Predictor`, which is a reference to the predictor for the corresponding object. Each `Predictor` consists of an array representing the methods associated with the object. There are `MAXmethods` entries in the array[11]. Each entry of the method array consists of an array of entries each of which is a <nextOID, weight> pair as in the previous strategies. In essence, in strategy 3, multiple instances (one per method) of the information used in strategy 2 are being maintained.

```
Predictor pred=GetPredictor(CurrOID);
OID nxt = OID of the next object actually accessed;
METHOD mthd=identifier of the invoked method on the
            current object;

int found=0;
for (int i=0; i<MAXentries; i++) {
   RefPredictorEntry p=pred.meth[mthd].entries[i];
   if (p->Oid==nxt){
        p->weight++;
        for (int j=i; j>0; j--) {
             //make the array ordered by the weights
             RefPredictorEntry x, y, tmp;
             x=pred.meth[mthd].entries[j-1];
             y=pred.meth[mthd].entries[j];
             if (x->weight < y->weight){ // swap
                  tmp->Oid=y->Oid;
```

[11] The compiler determines the number of methods per object class. The compiler is also responsible for determining the mapping between each method's name and its corresponding method number.

```
                    tmp->weight=y->weight;
                    y->Oid=x->Oid;
                    y->weight=x->weight;
                    x->Oid=tmp->Oid;
                    x->weight=tmp->weight;
               } else{
                    break;
               }
          }
          found=1;
     } else if (p->Oid!=-1){
          p->weight--;   // decrease weight on miss
          if (p->weight==0){
               p->Oid=-1;    // mark slot as free
          }

     }
}
if (found==0){ // next referenced object not found
// add next object's OID to array if space is left
   if (pred.meth[mthd].entries[MAXentries-1]->Oid
            == -1){
          // array is not full
          int Avg=0;
          for (i=0; i<MAXentries; i++) {
               RefPredictorEntry p=pred.meth[mthd].
                      entries[i];
               if (p->Oid == -1){  // found space
                    if (i!=0) {
                         Avg=Avg/i;
                         p->Oid=nxt;
                         p->weight=Avg;
                    } else {
                         p->Oid=nxt;
                         p->weight=MAXINT/2;
                    }
                    for (int j=i; j>0; j--) {
                         // reorder for new entry
                         RefPredictorEntry x, y, tmp;
                         x=pred.meth[mthd].
                              entries[j-1];
                         y=pred.meth[mthd].entries[j];
                         if (x->weight < y->weight){
                              tmp->Oid=y->Oid;
                              tmp->weight=y->weight;
                              y->Oid=x->Oid;
                              y->weight=x->weight;
                              x->Oid=tmp->Oid;
                              x->weight=tmp->weight;
                         } else{
                              break;
                         }
                    }
                    break;
```

```
                } else {
                    Avg=Avg+p->weight;
                }
            }
        }
    }
```

Figure 4.13 - Updating the Predictor Data Structure for Strategy 3

The code in Figure 4.13 illustrates the processing needed to maintain the weights associated with the objects that Oi has recently invoked methods on. Knowing the OID of the current object and the identifier of the method invoked on the object, the mechanism for updating the weights in this code is similar to that of the code for strategy 2 except that it updates the "next reference" information on a per-method rather than per-object basis. Thus, in the code, updates are made to `pred.meth[mthd].entries[j]` instead of to `pred.entries[j]`. As with strategy 2, the OIDs of all entries in the array maintained by the reference predictor are initially set to –1 and the code supports the dynamic addition of new entries in the reference predictor by locating an empty entry (which has an OID of –1). Again, the code dynamically updates the weights associated with the non-empty entries in the reference predictor in the same way as in strategy 2.

```
Predictor pred=GetPredictor(CurrOID);
METHOD mthd=id of the currently executing method;
RefPredictorEntry p;
int BestWeight;

// always pre-fetch the first object if it exists
p = pred.meth[mthd].entries[0];
if (p->Oid!=-1) {
   Pre-Fetch(p->Oid); // do the pre-fetch
   BestWeight=p->weight;
} else {
   exit; // no pre-fetching to be done yet
}

// pre-fetch other objects within ThresholdPctg
for (int i=1; i<MAXentries; i++) {
   p = pred.meth[mthd].entries[i];
   if (p->Oid==-1)
        break;
   if (p->weight >= (pred->ThresholdPctg*
        BestWeight)){
        Pre-Fetch(p->Oid); // do the pre-fetch
   else
        break;
}
```

Figure 4.14 - Using the Predictor for Strategy 3

Using the information in the reference predictor is straightforward and is illustrated in Figure 4.14. The only difference with the code for strategy 2 is that the object(s) to be pre-fetched are determined considering the current method (`mthd`) that is being executed. In this way, as described earlier, different objects may be selected for pre-fetching depending on which method is being executed. This, logically, should offer better pre-fetching results at the cost of storing more "next reference" information in the reference predictor.

4.5.4. Strategy 4: $\{PRE\text{-}O_i\}O_i.M_k \rightarrow \{O_j\}$

Strategy 4 predicts a set of objects that are likely to be accessed "next" given a method invocation path ("{PRE-Oi}Oi.Mk"). Using strategy 3 alone, the accuracy of object pre-fetching is still in question when the set of objects to be pre-fetched may change depending on how the current object method (Oi.Mk) is invoked. This may happen since an object method's behaviour can change depending on how it is invoked (i.e. based on what arguments it receives). Unfortunately, it is impractical to try to track and use "next reference" information for all possible combinations of input parameters since this would require vast amounts of storage as well as high runtime overhead. A rough predictor of likely change in object behaviour can be determined using the sequence of methods called to invoke the current object method. It seems logical that significantly different call sequences might lead to significantly different object behaviour (due to increased probability of different input parameters).

In strategy 4, the concept of path tracking will be incorporated into the prediction process to provide a basis for path-based pre-fetching. A "method invocation path" (or just "path" for short) is defined to be a sequence of method invocations made as part of a program's execution and "next access" information can be stored on a per-path basis to allow path-specific pre-fetching. The concept of a path as a part of a larger object method invocation sequence is illustrated in Figure 4.15. Naturally, this should produce more accurate results when there are significant differences in object method reference behavior depending on how the method is invoked.

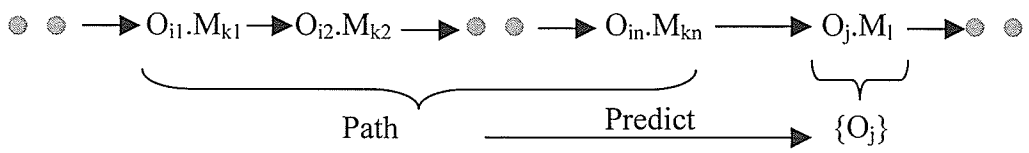

Figure 4.15 - The Concept of a Method Invocation Path

Data Structure for Strategy 4:

In strategy 4, the reference predictor for an object (Oi) stores a collection of possible "next" objects and their associated probabilities of reference (i.e. weights) for each possible method invocation path. Each reference predictor thus consists of an array representing the methods of the corresponding object and each entry in that array contains a pointer to a collection of recent invocation paths that each end with the corresponding method being called. For each path, there is then a pointer to an ordered array of two-element structures corresponding to the objects that are likely to be accessed next given that execution path. As before, the first element of each such structure stores the OID of the object likely to be pre-fetched and the second element stores the corresponding weight associated with that object. An example of the data structure used in the reference predictor is shown in Figure 4.16.

Object Prediction for Strategy 4:

When a method, M_k, on the current object, O_i, is invoked, the entry with OIDi will be looked up in the GDO. Based on the current path information, the reference predictor will be used to determine which objects are likely to be referenced next so they can be pre-fetched into local machine's memory.

A new issue with this strategy is how method invocation paths can be determined, how such path information can be represented, and how invocation paths can be tracked for use in pre-fetching. To solve this problem, a "path queue" can be used both to track the object method invocations as they are made during program execution and to store such information in the reference predictors. A hash function can also be used to reduce such paths to a more convenient representation for rapid comparison, etc.

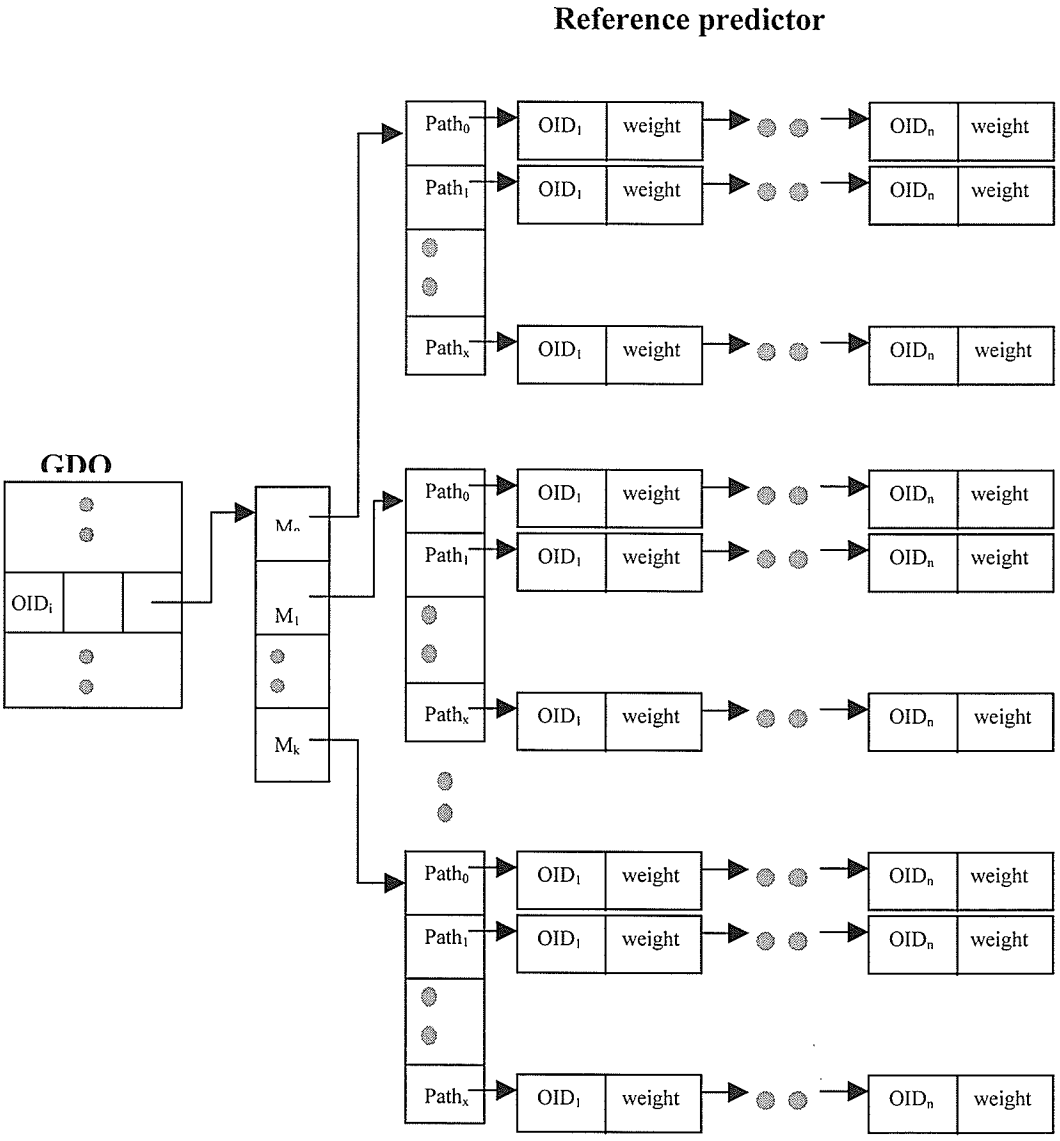

Figure 4.16 - Data structure for Strategy 4

Tracking Program Execution

An n-element "path queue" can be used to dynamically track the sequence consisting of the last n object method invocations made at any time. The value of n can be chosen based on how long a path is desired for use in pre-fetching. (Since exactly n object method identifiers will then be needed to determine a method invocation path.) Such a queue is illustrated in Figure 4.17. Each entry in the path queue contains an

<OID,MID> pair (which consists of the OID of the referenced object and the identifier[12] of the method invoked on it) and a pointer to the next record in the queue. The "Front" pointer points to the earliest record in the queue (i.e. the oldest method invocation in the path) and the "Rear" pointer points to the most recent record (and method invocation).

Logically, when a given method is executed on an object, the current path information queue can be matched against the information recorded in the path queues stored in the reference predictor for the current object to determine which prediction information should be used in pre-fetching. The *current* method invocation path must then be updated by removing the earliest record from the front of the queue and adding the current object method as a new record at the rear of the queue.

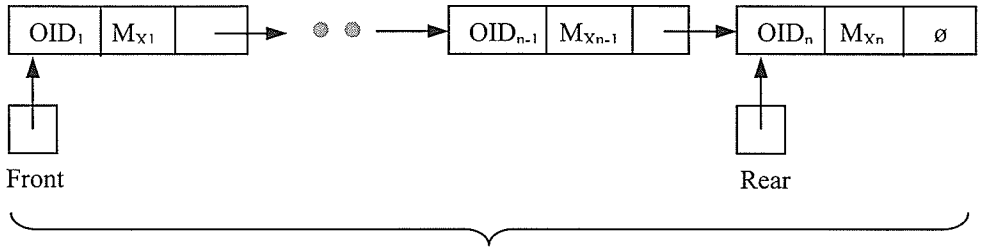

Figure 4.17 – Queue Structure for Path Tracking

Path hash function

Even given the numeric nature of the representation in each path queue element, searching on a *sequence* of object method invocations would be quite tedious. Accordingly, it is desirable to be able to map such a sequence to some simpler form for convenient manipulation. This can be done using hashing. The hashed forms of paths can then be used to rapidly distinguish between different paths (which will be a fundamental operation in strategy 4). Using this approach, each object method in the reference predictor will have a hash table associated with it for mapping from method invocation paths.

Since the requirements of a hash function in this situation are minimal, a simple division-based hashing method can be used to implement the hash function. In the

[12] The names of object methods are traditionally converted into method id numbers (MIDs) at compilation time (for example to index into a VTABLE in C++ implementations[16]).

division method for hashing, the size of the hash table, X, is selected to be a prime number. By making X a prime number, the likelihood that the keys will be evenly spread out across all the entries in the hash table is improved. Since, in the vast majority of cases, there are likely to be relatively few unique "paths" to any method, X can be selected to be a small prime number such as 13, 19, 23, 29, etc. As shown below, given pairs of the form < OIDi, M_k > (which denote method M_k on object O_i) the required hash function will be computed by multiplying the OID of object O_i by M_k+1[13] for every method referenced along the "path" and then adding up the results of those multiplications and taking the remainder on division by X. This value will, of course, be used to subscript into the hash table. Each entry in the hash table corresponds to a specific path leading to an execution of the current object method and references the objects that are candidates to be pre-fetched. Note that while hash collisions (i.e. two paths mapping to the same hash value) are unlikely, they are still possible. This is not an issue in this work since the correctness of the execution is not affected. Only the performance may, possibly, be affected since inaccurate pre-fetching may result. This is similar to the use of hashing on addresses in branch prediction tables as discussed in Section 2.2.4.

$$H(Queue) = (OID_1 \times (M_1+1) + OID_2 \times (M_2+1) + \ldots + OID_i \times (M_k+1)) \bmod X$$

Algorithms for strategy 4:

```
#define MAXentries "max number of next objects"
#define MAXmethods "max number of methods / object"
#define MAXpaths "max number of paths / method"

typedef struct rpentry {
   OID Oid=-1;     // initially no OID
   int weight=0;   // initially no weight
} *RefPredictorEntry;

typedef struct OnePa{
   RefPredictorEntry entries[MAXentries];
} *OnePath;

typedef struct MethPaths{
   OnePath path[MAXpaths];
} *MethodPaths;

typedef struct pred {
```

[13] M_k+1 must be used rather than M_k since method ids, unlike object ids, can be zero.

```
   MethodPaths meth[MAXmethods];
   float ThresholdPctg;     // value between 0 and 1
} *Predictor;

typedef struct queue{
   OID Oid;
   METHOD Method;
   Struct queue *next;
} *QueuePtr, *QueueFront, *QueueRear;
```

Figure 4.18 - The Predictor Data Structure for Strategy 4

As shown in Figure 4.18, each GDO entry will contain an entry of type `Predictor`, which is a reference to the predictor for the corresponding object. Each `Predictor` consists of an array storing elements corresponding to the methods associated with the object. Each entry in this array (`meth`) consists of an array whose elements implement the path queue through which the corresponding object method may be reached. The number of entries in the array is set to `MAXpaths`, which places an upper bound on the number of invocations, n, in a path. Each entry in the path array consists of the normal array of entries each of which is a <nextOID, weight> pair with OIDs as in strategies 2 and 3. For completeness, the data structure for the method invocation queue is also defined here. Each element in the queue contains an OID, the identifier of the method invoked on the object and a reference to the next element.

```
Predictor pred=GetPredictor(CurrOID);
OID nxt = OID of the next object actually accessed;
METHOD mthd=identifier of the invoked method on the
          current object;
int path=hash(CurrentPathQueue); // get path index

int found=0;
for (int i=0; i<MAXentries; i++) {
   RefPredictorEntry p=pred.meth[mthd].path[path].
                        entries[i];
   if (p->Oid==nxt){
        p->weight++;
        for (int j=i; j>0; j--) {
             //make the array ordered by the weights
             RefPredictorEntry x, y, tmp;
             x=pred.meth[mthd].path[path].entries[j-1];
             y=pred.meth[mthd].path[path].entries[j];
             if (x->weight < y->weight){ // swap
                  tmp->Oid=y->Oid;
                  tmp->weight=y->weight;
                  y->Oid=x->Oid;
                  y->weight=x->weight;
                  x->Oid=tmp->Oid;
```

```
                    x->weight=tmp->weight;
               } else{
                    break;
               }
          }
          found=1;
     } else if (p->Oid!=-1){
          p->weight--;   // decrease weight on miss
          if (p->weight==0){
               p->Oid=-1;    // mark slot as free
          }

     }
}

if (found==0){ // next referenced object not found
// add next object's OID to array if space is left
   if (pred.meth[mthd].path[path].entries[MAXentries-1]
               ->Oid == -1){
          // array is not full
          int Avg=0;
          for (i=0; i<MAXentries; i++) {
               RefPredictorEntry p=pred.meth[mthd].
                    path[path].entries[i];
               if (p->Oid == -1){  // found space
                    if (i!=0) {
                         Avg=Avg/i;
                         p->Oid=nxt;
                         p->weight=Avg;
                    } else {
                         p->Oid=nxt;
                         p->weight=MAXINT/2;
                    }
                    for (int j=i; j>0; j--) {
                         // reorder for new entry
                         RefPredictorEntry x, y, tmp;
                         x=pred.meth[mthd].path[path].
                              entries[j-1];
                         y=pred.meth[mthd].path[path].
                              entries[j];
                         if (x->weight < y->weight){
                              tmp->Oid=y->Oid;
                              tmp->weight=y->weight;
                              y->Oid=x->Oid;
                              y->weight=x->weight;
                              x->Oid=tmp->Oid;
                              x->weight=tmp->weight;
                              } else{
                                   break;
                              }
                    }
                    break;
               } else {
                    Avg=Avg+p->weight;
```

```
                }
            }
        }
    }
```

Figure 4.19 - Updating the Predictor Data Structure for Strategy 4

The code in Figure 4.19 illustrates the processing needed to maintain the weights associated with the objects that $O_i.M_k$ has recently invoked methods on when invoked at the end of the current method invocation path. This code is, naturally, similar to the code described in strategy 3 except that it maintains information on a per-path basis rather than on a per-method basis. Given the object and method identifiers for the current object method and the path number computed from the current path using the hash function, the selects the appropriate array of <nextOID, weight> pairs for updating. Note that the code for maintaining the *current* method execution path is independent of both updating and using the reference predictor. Maintaining this path information can be accomplished by straightforward augmentation of the code generated by the object/class compiler to report method invocations made to a runtime system that can build and update the current path. The code for updating the current invocation path is not explicitly shown in this thesis.

```
Predictor pred=GetPredictor(CurrOID);
METHOD mthd=id of the currently executing method;
int path=hash(CurrentPathQueue); // get path index
RefPredictorEntry p;
int BestWeight;

// always pre-fetch the first object if it exists
p = pred.meth[mthd].path[path].entries[0];
if (p->Oid!=-1) {
   Pre-Fetch(p->Oid); // do the pre-fetch
   BestWeight=p->weight;
} else {
   exit; // no pre-fetching to be done yet
}
// pre-fetch other objects within ThresholdPctg
for (int i=1; i<MAXentries; i++) {
   p = pred.meth[mthd].path[path].entries[i];
   if (p->Oid==-1)
        break;
   if (p->weight >= (pred->ThresholdPctg*
        BestWeight)){
        Pre-Fetch(p->Oid); // do the pre-fetch
   else
        break;
}
```

Figure 4.20 - Using the Predictor for Strategy 4

Again, using the information in the reference predictor is straightforward as illustrated in Figure 4.20. After determining the appropriate pre-fetch information using the current path information, we simply pre-fetch in the same way as in strategy 3.

4.5.5. Strategy 5: $O_i.M_k \rightarrow \{ \{O_{j1}\}, \{O_{j2}\}, \ldots \}$

Strategy 5 predicts a *sequence* of objects that are likely to be accessed "next", to a specified depth (as described in Section 4.4), given the current object method (but not, for the moment at least, considering the method invocation path used). Thus, strategy 5 extends strategy 3 (not strategy 4) by adding support for pre-fetching depths greater than one. This should improve pre-fetching effectiveness when some of the objects that are pre-fetched will execute for only a short period of time before invoking methods on other objects (at greater depth).

Data Structure for Strategy 5:

In this strategy, pre-fetching depth is added into the reference predictor to attempt to improve pre-fetching effectiveness. Recall that the pre-fetching depth is defined to be the "closeness" of a next object (e.g. Oj) to be referenced to the object currently being executed on (e.g. Oi). The depth to which pre-fetching is done is a tunable parameter and may be set to any appropriate value on a system-wide, per-object, or per-object method basis. To clearly and simply demonstrate the strategy, a fixed pre-fetching depth of three levels is assumed in this section.

To support pre-fetching depth, prediction information concerning which objects will be accessed next at (in this case) three levels must be recorded in the reference predictor. For each method, this information about object method invocations made in the past is collected and sorted by order of closeness in terms of the depth relationship to the current object. An invocation of a method on object (Oi), will then lead to the pre-fetching of next sets of objects at depth 1, depth 2 and depth 3. Thus in Strategy 5, the reference predictor for each object O_i will store a collection of possible "next" access information for each of the object methods. Each predictor will maintain an array representing the methods and each entry in the array will contain a reference to an ordered array for each of the possible pre-fetch depths (i.e. a 3 element array in the running example). For each depth, there is then a reference to an ordered array of two-

element structures corresponding to the objects that may be accessed next at the corresponding level for that object method. As always, the first element of each such structure stores the OID of the object likely to be pre-fetched and the second element stores the corresponding weight associated with that object. An example of the data structure used in the reference predictor is shown in Figure 4.21.

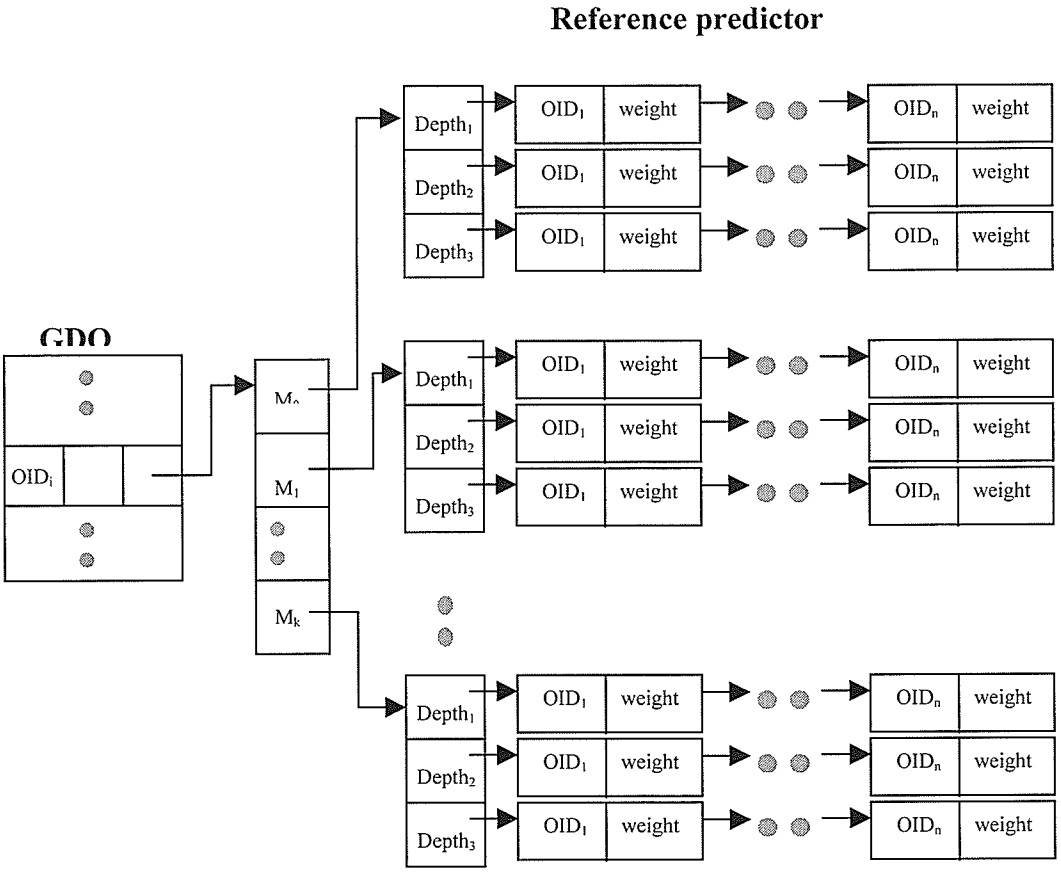

Figure 4.21 - Data structure for Strategy 5

Note that this implementation of pre-fetch depth makes no attempt to exploit the obvious relationship between an object that is invoked at depth `i` and one which is invoked by it at depth `i+1`. Thus, it may sometimes be that the objects fetched at depth `i+1` are not related to those fetched at depth `i`. While this may be undesirable since it may lead to decreased pre-fetching effectiveness, it does not affect the correctness of program execution and will therefore be tolerated. Further, there is a high probability that in many cases, the objects most likely to be accessed next at level `i+1` will be those that are referenced by the objects most likely to be accessed next at level `i`. (That is, in fact, why the objects at level `i+1` are likely to be accessed next.

Object Prediction for Strategy 5:

When method k of the current object O_i is invoked, OIDi will be looked up in the GDO and the reference predictor will be consulted to determine which sets of objects (from depths 1, 2 and 3) should be pre-fetched into the local machine's memory. At each level, it is possible to pre-fetch one or more objects using pre-fetch thresholds as described earlier. Thus, using strategy 5, the performance of pre-fetching can be improved by optimizing both the number of pre-fetching levels and the pre-fetching threshold (possibly independently at each level) to meet actual application needs.

Tracking Multiple Levels of Method Invocations

A new issue in this strategy is how the objects likely to be accessed next (up to depth three in the running example) can be determined. To solve this problem, a three-element queue is used to track the program execution by recording data on the last three object method invocations made. Such a queue is illustrated in Figure 4.22. This structure is, of course, exactly the same as the path queue described for use with strategy 4. The way that the information in the queue will be used, however, is very different.

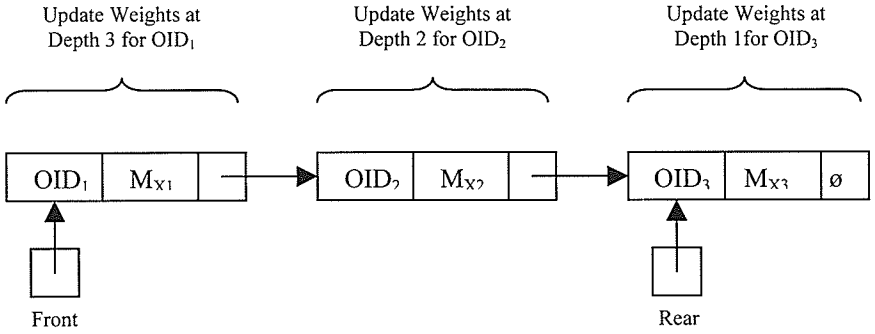

Figure 4.22 - Structure for Tracking Method Invocations

As in strategy 4, each element in the queue stores the OID of an object, the identifier of the method executed on it and a reference to the next element in the queue. The "Front" pointer points to the earliest record in the queue and the "Rear" pointer points to the most recent record. In strategy 5, when a method on the current object is executed, the information in the queue is used to identify the previous object method

invocations that lead to the current invocation. The reference predictor for each of these previous method invocations must be updated to reflect the fact that the current object method invocation followed it at a depth of 1, 2, or 3 (depending on whether the updated predictor corresponds to the third, second, or first entry in the queue). For example, if the current method invocation is $O_i.M_k$ and the three entries in the queue (from oldest to youngest) are $O_1.M_1$, $O_2.M_2$, and $O_3.M_3$, respectively then the depth 3 reference predictor corresponding for $O_1.M_1$, the depth 2 reference predictor corresponding for $O_2.M_2$, and the depth 1 reference predictor corresponding for $O_3.M_3$ have their weights for $O_i.M_k$ incremented. Following these weight updates, the queue of recent method invocations will be updated by removing the earliest record from the front of the queue and adding the object method just executed to the rear of the queue.

Algorithms for Strategy 5:

```
#define MAXentries "max number of next objects"
#define MAXmethods "max number of methods / object"
#define MAXdepths "max number of depth level"

typedef struct rpentry {
   OID Oid=-1;          // object identifier
   int weight=0;        // initially no weight
} *RefPredictorEntry;

typedef struct OneDep{
   RefPredictorEntry entries[MAXentries];
} *OneDepth;

typedef struct OneMeth{
   OneDepth depth[MAXdepths];
} *OneMethod;

typedef struct pred {
   OneMethod meth[MAXmethods];
   float ThresholdPctg;     // value between 0 and 1
} *Predictor;
```

Figure 4.23 - Data Structure for Strategy 5

As shown in Figure 4.23, each GDO entry will contain an entry of type `Predictor`, which is a reference to the predictor for the corresponding object. Each `Predictor` is similar to that in strategy 3 except that next access information is now maintained for multiple levels. Thus, each entry of the method array (`meth`) consists of an array with one entry for each of the supported depth levels. The number of entries in

the array is MAXdepths, which is assumed to be three for the description of strategy 5. Each entry in the depth array, naturally, consists of an array of entries each of which is a <nextOID, weight> pair. The data structure for the queue that is used to track the program execution is the same as in strategy 4 (shown in Figure 4.18).

```
QueuePtr=QueueFront; // Assumes the queue structure exists
for (int n=MAXdepths-1; n>=0; n--){
     Predictor pred=GetPredictor(QueuePtr->Oid);
     OID nxt=object identifier of next accessed object;
     METHOD mthd=the identifier of the invoked method;

     int found=0;
     for (int i=0; i<MAXentries; i++) {
          RefPredictorEntry p;
          p=pred.meth[QueuePtr->Method].depth[n].
               entries[i];
          if (p->Oid==nxt){
               p->weight++;
               for (int j=i; j>0; j--) {
                    //make the array ordered by the weights
                    RefPredictorEntry x, y, tmp;
                    x=pred.meth[QueuePtr->Method].depth[n].
                         entries[j-1];
                    y=pred.meth[QueuePtr->Method].depth[n].
                         entries[j];
                    if (x->weight < y->weight){ // swap
                         tmp->Oid=y->Oid;
                         tmp->weight=y->weight;
                         y->Oid=x->Oid;
                         y->weight=x->weight;
                         x->Oid=tmp->Oid;
                         x->weight=tmp->weight;
                    } else{
                         break;
                    }
               }
               found=1;
          } else if (p->Oid!=-1){
               p->weight--;   // decrease weight on miss
               if (p->weight==0){
                    p->Oid=-1;    // mark slot as free
               }

          }
     }

     if (found==0){ // next referenced object not found
        // add next object's OID to array if space is left
        if(pred.meth[QueuePtr->Method].depth[n].
               entries[MAXentries-1]->Oid==-1){
        // array is not full
        int Avg=0;
```

```
            for (i=0; i<MAXentries; i++) {
                  RefPredictorEntry p;
                p=pred.meth[QueuePtr->Method].depth[n].
                  entries[i];
                if (p->Oid == -1){  // found space
                  if (i!=0) {
                    Avg=Avg/i;
                    p->Oid=nxt;
                    p->weight=Avg;
                  } else {
                    p->Oid=nxt;
                    p->weight=MAXINT/2;
                  }
                  for (int j=i; j>0; j--) {
                  // reorder for new entry
                    RefPredictorEntry x, y, tmp;
                    x=pred.meth[QueuePtr->Method].
                      depth[n].entries[j-1];
                    y=pred.meth[QueuePtr->Method].
                      depth[n].entries[j];
                    if (x->weight < y->weight){
                      // swap
                      tmp->Oid=y->Oid;
                      tmp->weight=y->weight;
                      y->Oid=x->Oid;
                      y->weight=x->weight;
                      x->Oid=tmp->Oid;
                      x->weight=tmp->weight;
                    } else{
                      break;
                    }
                  }
                  break;
                } else {
                  Avg=Avg+p->weight;
                }
              }
            }
          }
          QueuePtr=QueuePtr->next;
}
```

Figure 4.24 - Updating the Predictor Data Structure for Strategy 5

The code in Figure 4.24 illustrates the processing needed to maintain the weights associated with the objects that have recently invoked methods that directly lead to the current invocation on Oi. The major difference between this algorithm and the one for strategy 3 is that the weights associated with a number of previous method invocations are updated (for the appropriate levels) instead of the weights associated with just one (the immediate predecessor of the current method). This is handled in the code by the

addition of the new (outer) `for` loop. Notice that the predictor is selected, in each iteration of that loop, based on the OID from the corresponding entry in the queue of recently executed object methods (`pred=GetPredictor(QueuePtr->Oid)`). Then, updates are made to the appropriate entries in the reference predictor data structure based on the iteration of the for loop (n) and the relevant object method invocation (`pred.meth[QueuePtr->Method].depth[n].entries[...]`).

```
Predictor pred=GetPredictor(CurrOID);
METHOD mthd=id of the currently executing method;
RefPredictorEntry p;
int BestWeight;

for(int n=0; n<MAXdepths; n++){
   // always pre-fetch the first object if it exists
   p = pred.meth[mthd].depth[n].entries[0];
   if (p->Oid!=-1) {
        Pre-Fetch(p->Oid); // do the pre-fetch
        BestWeight=p->weight;
   } else {
        exit; // no pre-fetching to be done yet
   }

   // pre-fetch other "next" objects which have
   // weights within ThresholdPctg of the first
   for (int i=1; i<MAXentries; i++) {
        p = pred.meth[mthd].depth[n].entries[i];
        if (p->Oid==-1)
             break;
        if (p->weight >= (pred->ThresholdPctg*
             BestWeight)){
        Pre-Fetch(p->Oid); // do the pre-fetch
        else
             break;
   }
}
```

Figure 4.25 - Using the Predictor for Strategy 5

The process of using the information in the reference predictor is illustrated in Figure 4.25. Note that a sequence (from each depth) of sets of objects is pre-fetched. This should handle the problems associated with pre-fetching objects on which short-duration methods will be executed.

4.5.6. Strategy 6: $\{PRE\text{-}O_i\}\ O_i.M_k \rightarrow \{\ \{O_{j1}\},\ \{O_{j2}\},\ \ldots\ \}$

Strategy 6 predicts a sequence of objects that are likely to be accessed "next", to a given depth, for a given method invocation path ("{PRE-Oi}Oi.Mk"). That is, strategy 6 combines the idea of path-based pre-fetching from strategy 4 with the concept of pre-fetch depth from strategy 5 (while maintaining support for pre-fetching threshold which is common to all algorithms from strategy 2 on). Strategy 6 should offer the best pre-fetching effectiveness by combining the concepts of both pre-fetching depth and path-based pre-fetching but at the highest storage and processing cost. To illustrate the idea as clearly as possible, a pre-fetching depth of three depth levels is again assumed.

Data Structure for Strategy 6:

In strategy 6, the concepts of path-based pre-fetching and pre-fetching depth are combined to improve probable pre-fetching efficiency. To clearly demonstrate the strategy, we again assume pre-fetching with 3 depth levels. Thus, an invocation of a method on object O_i will lead to the pre-fetching of sets of "next" objects at depth 1, depth 2 and depth 3. Further, the sets of objects will be potentially different given different method invocation paths leading to the method on object O_i. To support this strategy, information concerning both the possible method invocation paths and probable next object access patterns at depth of 3 must be recorded in the reference predictor.

The reference predictor for an object must store a collection of possible "next" objects and their associated probabilities to the specified pre-fetching depth that are specific to the methods of the current object and the possible execution paths to those methods. Thus, each reference predictor will consist of an array representing the methods where each entry in the array will contain a reference to another array of paths. For each path, there will be a reference to an array of depths each of which contains a reference to an ordered array of two-element structures corresponding to the objects that may be accessed next given the corresponding method invocation path and depth. As always, the first element of each such structure stores the OID of the object likely to be pre-fetched and the second element stores the corresponding weight associated with that object. An example of the data structure used in the reference predictor for strategy 6 is shown Figure 4.26.

Object Prediction for Strategy 6:

When a method, M_k, of the current object, O_i, is invoked, OIDi will, again, be looked up in the GDO. Based on the method invocation path that has lead to M_k's execution, the reference predictor will select the sets of objects that are likely to be accessed next to a depth of 3 and these objects will be pre-fetched into the local machine's memory.

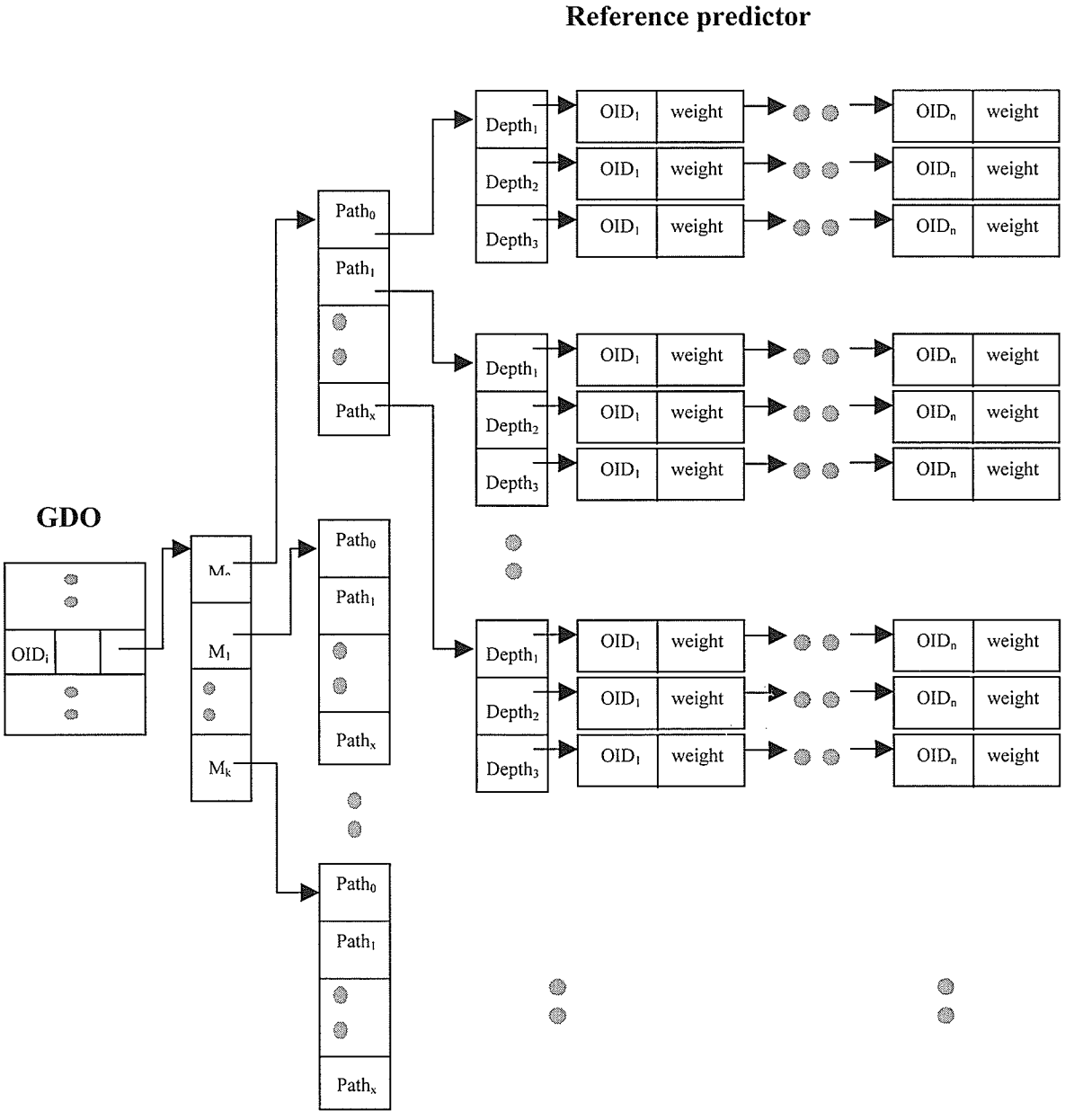

Figure 4.26 - Data structure for Strategy 6

Tracking Program Execution

In strategy 6, an n-element queue is used to track program execution (i.e. the most recent method invocations). This supports the use of n <OID, Method> pairs in path-based pre-fetching. The same queue is also used in the process of updating the weights of the sets of next objects that are likely to be accessed (to the prescribed pre-fetching depth of 3). Accordingly, n (the length of the stored path information) must be greater than or equal to 3 (the pre-fetch depth). The queue structure is illustrated in Figure 4.27.

As in strategy 4, each queue element includes the OID and method identifier of the corresponding object method invocation and a pointer to the next element in the queue. In addition, each queue element must also now contain a path number that identifies the path used to reach the method invocation described in that queue entry. As before, the "Front" pointer still points to the earliest record in the queue and the "Rear" pointer still points to the most recent record.

Again, assuming a pre-fetching depth of 3, once the next invoked object method is known, the system will first update the weights of the previously invoked objects to the supported depth level (3) using the last three records from the rear of the path queue. Second, the path queue itself is updated by removing the earliest record from the front of the queue and adding a record describing the new method invocation at the rear of the queue. Third, the path number is re-calculated by hashing on the object method information stored in the queue and the resulting path number is inserted into the "path" field of the new record.

Compared to previous strategies, strategy 6 should offer the best prediction in terms of accuracy and this should lead to improved overall system efficiency. Strategy 6 will be particularly useful in object systems with complex inter-object calling relationships (where the set of objects that should be pre-fetched may depend on the method invocation path to the current object method execution and where there may be multiple likely next objects and where some pre-fetched objects may have only short-lived methods executed on them).

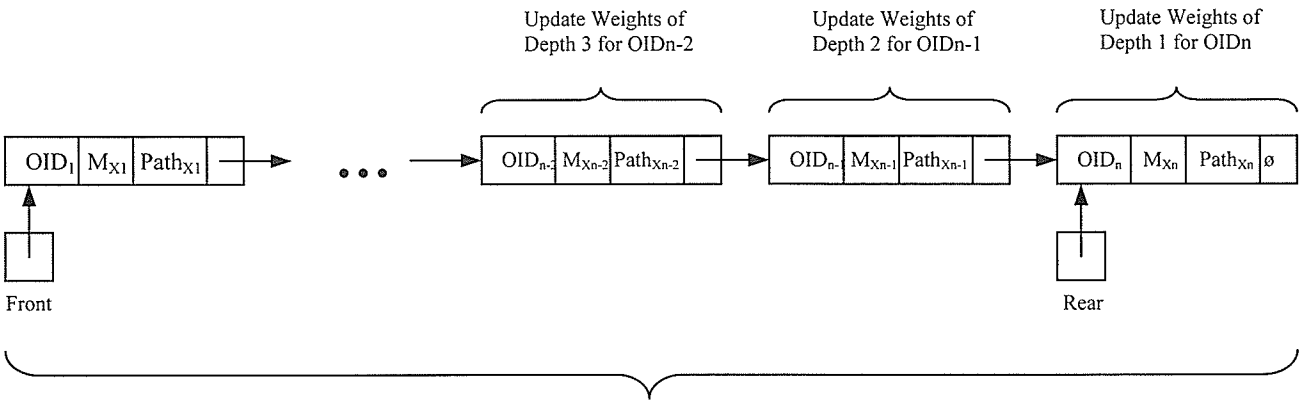

Figure 4.27 - Structure for Path and Depth Tracking

Algorithms for Strategy 6:

```
#define MAXentries "max number of next objects"
#define MAXmethods "max number of methods / object"
#define MAXpaths "max number of paths"
#define MAXdepths "max number of depth levels"

typedef struct rpentry {
   OID Oid=-1;          // object identifier
   int weight=0;        // initially no weight
} *RefPredictorEntry;

typedef struct OneDep{
   RefPredictorEntry entries[MAXentries];
} *OneDepth;

typedef struct OnePa{
   OneDepth depth[MAXdepths];
} *OnePath;

typedef struct OneMeth{
   OnePath path[MAXpaths];
} *OneMethod;

typedef struct pred {
   OneMethod meth[MAXmethods];
   float ThresholdPctg;     // value between 0 and 1
} *Predictor;

typedef struct queue{
   OID Oid;
   METHOD Method;
   int path;       // path to this method invocation
   Struct queue *next;
} *QueuePtr, *QueueFront, *QueueRear;
```

Figure 4.28 - The Predictor Data Structure for Strategy 6

As shown in Figure 4.28, each GDO entry will, again, contain an entry of type `Predictor`, which is a reference to the predictor for the corresponding object. Each `Predictor` consists of an array representing the methods associated with the corresponding object. The number of entries in this "method" array is set to `MAXmethods` and each entry of the array consists of an array representing the probable paths through which the object method may be reached. The number of entries in this array is set to `MAXpaths`, which can be determined based on specific application needs. Each entry in the path array consists of an array representing the different possible pre-fetching depth levels. The number of entries in the array is set to `MAXdepths`, which is, again, tunable based on application needs. Each entry in the depth array finally consists of an array of entries each of which is a <nextOID, weight> pair. The data structure for the queue that is used to track the program execution is also shown in Figure 4.28.

```
QueuePtr=QueueFront; // Assumes the queue structure exists
for (int n=MAXdepths-1; n>=0; n--){
     Predictor pred=GetPredictor(QueuePtr->Oid);
     OID nxt=object identifier of next accessed object;
     METHOD mthd=the identifier of the invoked method;

     int found=0;
     for (int i=0; i<MAXentries; i++) {
          RefPredictorEntry p;
          p=pred.meth[QueuePtr->Method].
               path[QueuePtr>path].depth[n].entries[i];
          if (p->Oid==nxt){
               p->weight++;
               for (int j=i; j>0; j--) {
                    //make the array ordered by the weights
                    RefPredictorEntry x, y, tmp;
                    x=pred.meth[QueuePtr->Method].
                         path[QueuePtr->path].depth[n].
                         entries[j-1];
                    y=pred.meth[QueuePtr->Method].
                         path[QueuePtr->path].depth[n].
                         entries[j];
                    if (x->weight < y->weight){ // swap
                         tmp->Oid=y->Oid;
                         tmp->weight=y->weight;
                         y->Oid=x->Oid;
                         y->weight=x->weight;
                         x->Oid=tmp->Oid;
                         x->weight=tmp->weight;
                    } else{
                         break;
                    }
               }
```

```
            found=1;
        } else if (p->Oid!=-1){
            p->weight--;   // decrease weight on miss
            if (p->weight==0){
                p->Oid=-1;    // mark slot as free
            }

        }
}

if (found==0){ // next referenced object not found
   // add next object's OID to array if space is left
   if(pred.meth[QueuePtr->Method].path[QueuePtr->path].
        depth[n].entries[MAXentries-1]->Oid==-1){
      // array is not full
      int Avg=0;
      for (i=0; i<MAXentries; i++) {
            RefPredictorEntry p;
            p=pred.meth[QueuePtr->Method].
                path[QueuePtr->path].depth[n].
                entries[i];
            if (p->Oid == -1){  // found space
                if (i!=0) {
                    Avg=Avg/i;
                    p->Oid=nxt;
                    p->weight=Avg;
                } else {
                    p->Oid=nxt;
                    p->weight=MAXINT/2;
                }
                for (int j=i; j>0; j--) {
                    // reorder for new entry
                    RefPredictorEntry x, y, tmp;
                    x=pred.meth[QueuePtr->Method].
                        path[QueuePtr->path].
                        depth[n].entries[j-1];
                    y=pred.meth[QueuePtr->Method].
                        path[QueuePtr->path].
                        depth[n].entries[j];
                    if (x->weight < y->weight){
                        // swap
                        tmp->Oid=y->Oid;
                        tmp->weight=y->weight;
                        y->Oid=x->Oid;
                        y->weight=x->weight;
                        x->Oid=tmp->Oid;
                        x->weight=tmp->weight;
                    } else{
                        break;
                    }
                }
                break;
            } else {
                Avg=Avg+p->weight;
```

```
                }
            }
        }
    }
    QueuePtr=QueuePtr->next;
}
```

Figure 4.29 - Updating the Predictor Data Structure for Strategy 6

The code in Figure 4.29 illustrates the processing needed to maintain the weights stored by the reference predictor. Given the OID and method identifier of the current object method as well as the path taken to reach the object method and what the pre-fetching depth is, the code updates the weights associated with the three (in the running example) object methods invoked immediately preceding the current object method. As in previous methods, the reference predictor dynamically updates the stored object method access information.

```
Predictor pred=GetPredictor(CurrOID);
METHOD mthd=id of the currently executing method;
int path=hash(CurrentPathQueue); // get path index
RefPredictorEntry p;
int BestWeight;

for(int n=0; n<MAXdepths; n++){
   // always pre-fetch the first object if it exists
   p = pred.meth[mthd].path[path].depth[n].
        entries[0];
   if (p->Oid!=-1) {
        Pre-Fetch(p->Oid); // do the pre-fetch
        BestWeight=p->weight;
   } else {
        exit; // no pre-fetching to be done yet
   }

   // pre-fetch other "next" objects which have
   // weights within ThresholdPctg of the first
   for (int i=1; i<MAXentries; i++) {
        p = pred.meth[mthd].path[path].depth[n].
             entries[i];
        if (p->Oid==-1)
             break;
        if (p->weight >= (pred->ThresholdPctg*
             BestWeight)){
             Pre-Fetch(p->Oid); // do the pre-fetch
        else
             break;
   }
}
```

Figure 4.30 - Using the Predictor for Strategy 6

The use of the information stored in the reference predictor is illustrated in Figure 4.30. This code is very similar to that of strategy 5 except that it exploits the additional path information stored for strategy 6 ("`.path[path]`" in the code above).

5. Assessment of the Algorithms

While it is not possible to do truly meaningful simulations of the pre-fetching strategies proposed in the preceding chapter, due to a lack of real-world data on inter-object access patterns in persistent object systems, it is important to try to assess under what conditions the various strategies (and the algorithms implementing them) would be likely to be effective. This assessment must factor in the costs of the algorithms as well as certain characteristics of the objects being pre-fetched.

5.1. Applicability of the Proposed Techniques

In the previous chapter, six strategies for pre-fetching objects in a persistent distributed object system were proposed. Each uses a "reference predictor" that is associated with the relevant object's entry in the Global Directory of Objects (which describes all the objects in the system). Each strategy builds on the previous strategy(s) to attempt to enhance the effectiveness of pre-fetching to deal with specific inter-object access characteristics. In this section, the key features of each strategy are summarized and a discussion of each strategy's likely applicability is provided.

5.1.1. Strategy 1: $O_i \rightarrow O_j$

Using strategy 1, the reference predictor has knowledge of the current object, O_i, alone (not the method being executed on it) and only the single object having the highest weight, O_j, will be pre-fetched. This strategy is easy to implement and the total cost of the pre-fetching (i.e. the cost for the run-time computation done by the predictor and the cost for transferring the single predicted object together with its reference predictor over the network) is low. Unfortunately, this strategy works well only when there is a consistent relationship between the current object (regardless of which method is being executed on it) and a single object that will always be accessed next. Since this sort of relationship is unlikely to be seen in practice, strategy 1 represents little more than a baseline strategy that is useful as a starting point for implementing, and for comparing with, the other strategies. In a more complicated (and realistic) object system, the access patterns

between objects will likely be far more variable and therefore pre-fetching only the single candidate object with the highest weight would likely lead to a high miss rate and corresponding performance penalty. Given a situation where one object invokes methods on one of a number of different objects with roughly equal probability the pre-fetching provided by strategy 1 will not be effective. A real-world example where this sort of situation might occur is in a simulation program. In a simple simulation application, the simulation manager object might invoke methods on other objects (representing the components in the simulation) in a round-robin fashion. The manager object does not consistently invoke methods on a single, specific next object. Thus, strategy 1 will perform poorly. This is, of course, only one of many possible scenarios where strategy 1 would fail to be effective.

5.1.2. Strategy 2: $o_i \rightarrow \{ o_j \}$

In strategy 2, the reference predictor again has knowledge of the current object, O_i, but not the method being executed on it. Given this information, a *set* of objects (say a fixed number N having the largest weights or, perhaps, all those with weights exceeding a threshold) will be pre-fetched. By using this strategy, the problem from strategy 1 that was just described can be solved since more than one object can be pre-fetched. By pre-fetching more than one object, the probability that the correct object will have been pre-fetched into local machine's memory before the system needs it is increased. Thus, in the object-based simulation example described earlier, several objects (all of which will eventually be needed) will be pre-fetched using strategy 2 thereby increasing overall system performance.

A critical issue in the use of strategy 2 is determining how many objects to pre-fetch. As described earlier, this might be determined using a fixed number or via some sort of threshold (i.e. pre-fetch if the weight is within a percentage of the greatest weight[14]). Depending on the individual application, there may be many objects that would be useful to pre-fetch or not many at all (e.g. there might be a very large number of simulation objects). A related issue is how many entries in the array of "next objects" should be supported since this places a bound on the number of objects that can be pre-

[14] This option is what was actually described in strategies 2 through 6.

fetched. There are, of course, several side-effects to be considered as well including the potential for cache-pollution if the prediction strategy is inaccurate as well as the cost of storing many entries in the reference predictor (and transferring them as the object migrates from site to site). It certainly makes sense to set the pre-fetching threshold value on a per-object basis (as was done beginning with strategy2) to allow for different pre-fetching behaviour corresponding to different object behaviour. Exactly how to determine what this threshold value should be for each object, however, is an issue that is beyond the scope of this thesis and is therefore not discussed.

The cost of pre-fetching is increased using strategy 2 due to the potential need to pre-fetch more objects from remote sites (the cost of selecting the objects to pre-fetch does not increased significantly). This cost must be offset by improved performance. Strategy 2 is clearly an improvement over strategy 1 but the real issue, however, is how well it will work generally. Strategy 2 will perform well if the current object, O_i, has only a few dominant[15] methods, which invoke methods on a consistent set of other objects. If there are many dominant methods then there will be a greater probability of variance in which objects are accessed by different methods. Further, if the set of next objects is not consistent, mis-predictions will occur. In both cases, strategy 2 will be less effective than it might be. There are many situations that occur in practice where there may be many dominant methods for a single object. For example, a user-interface object that allows a user to interactively select from one of a number of functions, will invoke different methods on a "core application object" depending on which function is selected. If the user frequently selects different functions, each of the methods corresponding to those functions will be frequently executed on the core application object. Further, each of those methods will likely invoke methods on different sets of objects because they perform distinct functions. Thus, predicting a single set of objects that will be "accessed next" after the core application object will be impossible.

The chief problem with strategy 2 is that the predictions it makes are not method specific (i.e. it predicts a single set of "next" objects regardless of which method is executing on the current object, O_i). As in the example just described, there may be

[15] In this context "dominant" means frequently accessed.

significant variance in the set of "next accessed" objects based on which method is executed on the current object.

5.1.3. Strategy 3: $O_i.M_k \rightarrow \{O_j\}$

Strategy 3 is method invocation specific. That is, both the current object and method identifiers are used to select a set of objects (with the greatest weights) to be pre-fetched. Strategy 3 will, thus, improve the accuracy of pre-fetching since different sets of objects can be pre-fetched for different current method invocations. In particular, this will allow for effective pre-fetching when the current object (like the "core application object" described previously) has multiple dominant methods that make invocations on largely disjoint sets of objects. The primary increased cost associated with using strategy 3 is the need to store significantly more information in the reference predictor. If "next access" information is maintained for k methods, then the storage cost of strategy 3 will be k times the cost for strategy 2. Generally speaking, storage is cheap, but all the data that is stored in the reference predictor must also be transferred with the corresponding object when it is pre-fetched to a new site. Fortunately, in practice, few objects are likely to have a truly large number of dominant methods since these objects would then violate the simplicity in design principles (i.e. k will be small). As a result, the added cost for transmitting the reference predictors for strategy 3 will not be large.

Strategy 3 is probably the first pre-fetching strategy of the six presented in the thesis that could potentially be used in practice. It exploits, however, only one of the three pre-fetching optimizations discussed in Section 4.4. Thus, there should be, and are, inter-object access patterns that may occur in practice that strategy 3 fails to handle well.

Strategy 3 will fail to pre-fetch effectively when the set of objects accessed following a method invocation on the current object may change from invocation to invocation. This will happen whenever the current object method's invocation behaviour changes dynamically. Such changes may occur due to differences in data entered by the user (which cannot be predicted) or due to differences in method input parameters that can depend (at least in part) on "how" a method is invoked. How a method is invoked can be determined by analyzing the preceding sequence of method invocations (i.e. the "method invocation path") that lead to the current method invocation. (Consider a

method that may be invoked by one object method with a parameter value of X and by a different object method with a parameter value of Y.) Clearly, selecting objects to be pre-fetched based only on the current object method (as is done by strategy 3) may therefore be insufficient. By also considering the method invocation path used to reach the current method (i.e. path-based pre-fetching as is done in strategy 4), the accuracy of pre-fetching may be further improved. A practical example where path-based pre-fetching could provide improved pre-fetching accuracy might be in an object-oriented CAD (Computer Aided Design) environment. In such an environment, it is common to have to do certain design rule checks. Further, in such an environment, objects typically "own" several subordinate objects (e.g. an "airplane" object might own its "fuselage", "cockpit", "tail", and both "wing" objects). Different design rule checks might have to be performed on the same component depending on which owning component is requesting the check. For example, an engine turbine might be tested for stress during operation as a part of the engine and for weight constraints as a part of the wing it resides on. The behaviour of the design rule checking method for the "turbine" object would therefore have to vary depending on whether it was invoked by the "engine" or the "wing" object (and call appropriate but potentially different methods accordingly).

Strategy 3 will also fail to be effective when the objects it selects for pre-fetching are used only briefly because the method invocations made on them are short-lived. In this case, there will be insufficient time to pre-fetch the next set of objects to be accessed. By pre-fetching not just the next "level" of objects (i.e. those on which methods are invoked directly by the current object method) but also one or more subsequent levels (those on which methods are invoked by the methods invoked by the current object method, etc.) this problem can be avoided. Doing this is exploiting a pre-fetch depth greater than one. An example where pre-fetching more than one level of objects is likely to be necessary is when dealing with an object-oriented application that uses a façade[16] to abstract away differences in system interfaces. The methods on the façade object(s) are typically extremely short-lived (as they commonly just do simple argument transformations) and it will therefore be necessary to also pre-fetch the objects the façade

[16] A façade is a classis object oriented design pattern. The interested reader is referred to [18] for more information.

refers to in order to achieve effective pre-fetching and thereby improved performance. Strategy 5 applies pre-fetching depth to address this problem.

5.1.4. Strategy 4: $\{PRE\text{-}O_i\}O_i.M_k \rightarrow \{O_j\}$

Strategy 4 exploits path-based pre-fetching so that a set of objects will be pre-fetched depending on the method invocation path ("{PRE-O_i}") leading to the current object method invocation ("O_i M_k"). As described, this strategy should further improve the accuracy of object pre-fetching over strategy 3 if there are significant differences in object method reference behavior based on calling path.

A serious issue with the use of strategy 4 is that it further increases the size of the data that must be stored by the reference predictor (over that required by strategy 3). Unfortunately, if there are many potential paths that may be taken to reach a given object method, then this increase in size may be a problem. In general, if there are p paths to each of k methods then strategy 4 will require p times more storage than strategy 3 and p*k times more storage than strategy 2. Further, all of these bytes will have to migrate with the corresponding objects. In slow networks, this could cripple overall system performance rather than improve it. Clearly, the size of p must be limited to ensure reasonable performance. In most cases, it is likely that the number of paths to a given object method will be small. In a very large distributed persistent object system, however, this may not always be true. Limiting p only by the expected available network bandwidth would be an ideal solution but predicting available network bandwidth is beyond the scope of this thesis and will not be considered.

Strategy 4 does not incur a significant amount of additional overhead in selecting the objects to pre-fetch. This is because of the use of hash function, which reduces a complex path specification to a simple integer that can be used as a subscript into the array of paths described in Section 4.5.4. A small, added cost during object access is incurred when using strategy 4 to keep track of the most recent object method invocations to form the current execution path. Unless objects will be extremely short lived (which is not the common case in distributed persistent object systems) this overhead should not be a serious issue.

As with strategy 3, strategy 4 only pre-fetches the "next" objects to be accessed to a depth of 1 so it may provide only limited benefit if the program execution will require objects in subsequent levels very quickly.

5.1.5. Strategy 5: $O_i.M_k \rightarrow \{ \{O_{j1}\}, \{O_{j2}\}, \ldots \}$

Strategy 5 adds pre-fetching depth to strategy 3. Thus, a set of objects at multiple depths (from the current object method) that are likely to be accessed next will be pre-fetched. As described, this will enhance pre-fetching effectiveness when some objects to be pre-fetched will execute for only a very short period of time.

Strategy 5 requires more storage than strategy 3 but not significantly more since it is unlikely that large pre-fetch depth values would be needed.[17] Strategy 5 will pre-fetch many more objects into the local machine's memory than strategy 3. The question is: " how likely are those objects to be needed in the future". Strategy 5 only makes sense of the objects pre-fetched are those that actually do follow the flow of program execution. Otherwise, it may waste system resources by consuming network bandwidth for object transmission and by causing pollution of the memory cache. This makes the selection of the pre-fetch depth parameter for each object very important.

As with strategy 4, there is also a run-time overhead for maintaining a history of recent object method invocations to permit updating weight information for previous methods within the current depth range (as described in Section 4.5.5).

While strategy 5 does incorporate pre-fetching depth, it does not pre-fetch differently based on object-invocation path. Thus, it may provide limited benefit when there is variance in object method behaviour based on method invocation path.

5.1.6. Strategy 6: $\{\text{PRE-}O_i\}\ O_i.M_k \rightarrow \{ \{O_{j1}\}, \{O_{j2}\}, \ldots \}$

Strategy 6 combines the added benefits offered by strategy 4 (path-based pre-fetching) and strategy 5 (pre-fetching depth) and, accordingly, overcomes all the problems raised with earlier strategies but also, being the most complicated strategy, incurs all the added costs of the other strategies. As with strategy 4, the size of the

[17] In practice, it would be uncommon to have *many* layers of objects with minimal processing since systems designed in this way tend to have inadequate performance even in non-distributed environments.

reference predictor data structures (and associated cost of sending them over the network) are the chief concerns with using strategy 6.

5.2. Implementation Decisions and Performance

This section discusses the motivation behind certain general implementation decisions that were applied to all the algorithms. These decisions can be seen reflected in the code presented in the preceding chapter.

One fundamental design decision was to minimize the use of pointers whenever possible. While the use of dynamic (pointer-based) data structures does allow for flexibility in algorithms it often also exacts a significant performance penalty in terms of both pointer chasing and poor cache performance due to decreased locality of reference. Because of the need to pre-fetch in a timely fashion, array structures were favoured over alternatives such as linked lists. Selecting arrays over lists was views as a particularly critical issue for the latter strategies presented where the size of the data structures that must be manipulated may become large enough to have a significant performance impact. The reader should also note that although some pointer variables were used for convenience in the data structures presented they too could be removed by simply embedding certain array-based structures within other array-based structures rather than referring to them using pointers/references.

For similar reasons, it was decided that it was preferable to place fixed bounds on such things as the number of "next referenced objects" that could be supported. In addition to being a requirement for the use of the more efficient array implementations, it was felt that having fixed limits would have virtually no negative side effects since, in most cases, the number of entries provided should be adequate to permit efficient and effective pre-fetching for the vast majority of objects that are likely to occur in practice. (For example, it would be highly unlikely to find a real object method that would make more than a handful of invocations on the methods of other objects with approximately the same, high, probability.) Past experience in developing software in almost all application areas has shown that it is a mistake to try to make code highly efficient for all possible scenarios. It is far more effective to "make the common case fast" and simply make the other cases "correct".

There is another issue that will factor into determining the overall efficiency of all the pre-fetching strategies discussed which has nothing to do with the code presented. Whenever an object method is invoked, the reference predictor corresponding to that object must be used to predict and pre-fetch those objects that are likely to be accessed next. As a result, the code in the reference predictors must, naturally, be efficient (this is why fixed size arrays, hashing, etc. were used). Further, to use the reference predictors they must be resident in the local machine's memory. Each predictor, however, is attached to the corresponding object's GDO entry and the GDO is a distributed persistent data structure. If the reference predictor is not available at the machine where the corresponding object is being accessed, it is highly unlikely that timely pre-fetching can be done. Thus, it is absolutely necessary that reference predictors be fetched together with the objects they apply to. This is not difficult to do since the GDO must be consulted to locate each object anyway and the corresponding reference predictor can be transferred when this is done. Of course, as soon as a copy of the predictor is taken, there is a consistency issue. Fortunately, the reference predictor will never be updated by more than one machine at a time (the one where the corresponding object is being accessed) and thus this is not an issue.[18]

A final performance related issue (and an, as yet, unstated assumption) is that the system will not attempt to pre-fetch objects that are already memory resident. Again, implementing this should be straightforward to implement.

5.3. Efficiency of the Algorithms

The algorithms for all of the strategies are efficient and scalable. In this section, an attempt is made to quantify how many resources each of the proposed pre-fetching strategies will require. In analyzing the resource consumption of an algorithm, the most important issue is generally its running time. Although several factors affect the absolute running time of a program (including the compiler, operating system and computer used) these are beyond the scope of this thesis. Thus, this section considers only the running time of the algorithms themselves.

---

[18] This assumes that updates to the predictor data structure are protected from inadvertent loss due to system failures.

For each strategy, there are two corresponding algorithms: one is for updating the data structure which records the history of the access patterns and the other is for actually using the information recorded in the data structure to do the pre-fetching. It is important to realize that, ultimately, each algorithm for each strategy deals only with an array of `MAXentries` elements describing potential next objects and their corresponding weights. In the more advanced strategies, the "correct" array to consider is selected first by a simple sequence of one or more subscripting operations. (For example, in strategy 3, the correct array is selected by indexing using the current method identifier to index into an array of methods, while in strategy 4, the correct array is selected by hashing the current path to an index into an array of paths and then using the method identifier to index into an array of methods, etc.) None of the subscripting operations contributes more than a fixed constant time (i.e. O(1)) to the running time of any of the algorithms.

In strategy 1 updating the stored weights in the reference predictor will take O(`MAXentries`) time. This is required to both update the weights (increasing the weight of the next object referenced and decreasing all the others) and to manage the collection of <nextOID, weight> pairs (i.e. to handle inserting new entries, etc.). Actually doing the pre-fetching also requires O(`MAXentries`) time since the array is maintained in unsorted order and the algorithm must search for the entry with the largest weight.[19]

In strategy 2 updating the stored weights will take O(`MAXentries`^{2}) time because the entries in the array may have to be shuffled after changes are made to the weight field in the selected element. Normally, this will require little or no shuffling but in the worst case an entire second pass over the array may be required. Thus, the *expected* execution time is actually linear not quadratic in `MAXentries`. Doing the pre-fetching requires O(`MAXentries`) time even though the array is now sorted. This is because pre-fetching threshold is supported in strategy 2 and, in the worst case, all the entries in the array may be within `ThresholdPctg` of the largest weight thereby requiring a complete scan over all array entries.

The worst-case running times of the algorithms for strategies 3 through 6 are identical to those for strategy 2 since all that changes between the algorithms is how the

[19] Of course, it would also be possible to maintain the array in sorted order but this would be done at the cost of increased running time when updating the data in the reference predictor.

"correct" array of <nextOID, weight> pairs is selected and this is, as described earlier, an O(1) process. Strategies 4, 5, and 6 all require that the runtime environment of the distributed persistent object system track the most recent object method invocations. In general, the number of such invocations tracked will be a small constant and the overhead of tracking them will be insignificant.

5.4. Expected Time for Pre-Fetching

To assess the practicality of the pre-fetching strategies presented, it is important to have some sense for how long it will take to actually migrate an object (and its associated reference descriptor) from a remote node in response to a pre-fetch request. The dominant factors in this process are, of course, the speed of the network and the size of the data to be transferred. In this section, some simple computations of the time required for pre-fetching objects are presented in tabular form and some comments are made about the likely applicability of the pre-fetching techniques for different kinds of commercial networks and different sizes of objects.

To provide a concrete example, a distributed persistent object system built on top of traditional (10mbps), fast (100mbps) and Giga (1000mbps) Ethernet is considered. Fast Ethernet is now the dominant commodity LAN technology but there is still a lot of traditional Ethernet in use and a good deal of gigabit Ethernet has also been deployed. Thus, this seems to be a reasonable assumption to make.

Numerous assumptions concerning the objects to be pre-fetched must also be made. In particular, on average, it is assumed that each method in an object would be likely to invoke no more than 10 methods on other objects. Thus, `MAXentries` is assumed to be 10 and this (the maximum) is the assumed number of objects that will be pre-fetched in strategies 2, 3 and 4. Where appropriate, a pre-fetching depth of 3 levels is also assumed. Therefore, the maximum number of objects to be pre-fetched in schemes incorporating pre-fetching depth would be 30. The object size is assumed to be 4K bytes. Many objects (together with their reference predictors) will be both larger and smaller than 4K. The transfer of smaller objects would be unlikely though since for convenience and efficiency reasons, transfers in a DSVM (the assumed execution environment) would normally be done in multiples of the physical page size (4K bytes is a small but realistic page size).

Given an object size of 4K, strategy 1 will transfer 4K bytes of data, strategies 2, 3 and 4 will transfer 4x10=40K bytes of data and strategies 5 and 6 (which support pre-fetching depth) will transfer 3x40=120K bytes of data.

The expected times for pre-fetching in each of the 6 strategies were calculated and are shown in Table 5.1. Expected times are shown for each method and for each of the three different types of Ethernet networks conservatively assuming that only between 20% and 40% of the network bandwidth will be available[20].

Pre-fetching strategies	# of objects to be pre-fetched	Total size (Bytes)	Expected time for the pre-fetching (ms)					
			10Mbps Ethernet		100Mbps Ethernet		1Gbps Ethernet	
			20%	40%	20%	40%	20%	40%
Strategy 1	1	4K	16	8	1.6	0.8	0.16	0.08
Strategy 2	10	40K	160	80	16	8	1.6	0.8
Strategy 3	10	40K	160	80	16	8	1.6	0.8
Strategy 4	10	40K	160	80	16	8	1.6	0.8
Strategy 5	30	120K	480	240	48	24	4.8	2.4
Strategy 6	30	120K	480	240	48	24	4.8	2.4

Table 5.1 - Expected Time for Pre-fetching

To be effective, pre-fetched objects must reach the local machine before they are required. This means that the transfer times shown above must be less than the execution time of the object method for which the pre-fetch was issued (except when pre-fetching multiple levels of objects in which case there may be additional time). Assuming that a processor can execute on the order of 10^9 instructions per second (via a combination of high clock frequency, superscalar execution, etc.) then 10^6 instructions can be executed per millisecond. Of the total instructions executed under a conventional multi-programmed operating system supporting several users, perhaps as much as 5% might be allocated to any user in a given time period. Thus, perhaps 50K instructions might be executed for any given user at any time (ignoring I/O and other delays).

Examining Table 5.1 it is clear that network speed is a crucial issue in making pre-fetching effective. Except for the highest speed networks and ignoring start-up latency, object transfers will take on the order of 10 milliseconds. To "hide" the delay associated with object loading, the currently executing object method will have to be quite compute-intensive. Of course, those methods that perform operations (I/O, inter-process

[20] This assumption is made to account for protocol overhead and the potential used of shared segments where contention may occur. With dedicated switched networks significantly more bandwidth would probably be available.

communications, etc.) that will cause them to block will be less likely to suffer from late arrival of pre-fetched objects. Based on the numbers presented in the table, it seems that traditional (10mbps) Ethernet is inappropriate for supporting pre-fetching in a distributed persistent object system (and is, in fact, also inappropriate for supporting such a system generally).

From the data in the table, it also appears that strategies 5 and 6 would be inappropriate except for Gigabit Ethernet. This, however, is not the case. The data in the table simply computes the transfer time based on the number of bytes transferred. This gives a misleading impression. When pre-fetching depth is used, when an object method $O_i.M_k$ is executed, several levels of "next" objects will be pre-fetched. When the next object method is executed (say O_jM_l) many of the objects specified to be pre-fetched for it will either already have been pre-fetched or be in the process of being pre-fetched. This means they do not have to be pre-fetched again and this decreases the *actual* bandwidth required for pre-fetching. It should also be noted that using pre-fetch depth, even if the first level(s) of pre-fetched objects arrive late, subsequent levels are very likely to be pre-fetched successfully. Once successful pre-fetching has occurred, it is possible to speculatively pre-fetch other levels of objects so that, in a pipeline-like fashion, objects are always available before they are needed[21]. Thus, selecting an appropriate pre-fetch depth can be used to deal with network latencies. Even so, assuming conventional IP-based networking, it is likely that pre-fetching (and object shipping itself) will only be practical in a local-area network context.

It is probably safe to conclude that using a technique that includes pre-fetching depth is very important to successful pre-fetching. Correspondingly, and considering the increased cost of storing next-reference data for many paths, it seems likely that strategy 5 will be the most practical pre-fetching technique in practice.

Note that the conclusions presented from Table 5.1 are specific to the assumptions made when generating the table. Different conclusions can be drawn if those assumptions are inaccurate. Particularly, if the duration of object methods can be shown to be high and/or if the multi-programming degree is large, then pre-fetching and DSVM-based

[21] With the rapidly decreasing cost of memory, this is an even more attractive option since the potential danger of memory cache pollution can be offset by providing larger (cheap) main memories.

systems themselves become more widely applicable even with older network technologies. Additionally, the assumptions concerning the limited available bandwidth in Ethernet networks was quite conservative.

6.Conclusions and Future Work

6.1. Conclusions

There is a need to find techniques to improve system performance in persistent distributed object systems. A chief problem in such systems is the relatively high network latency associated with migrating objects to where they are needed. One way to try to hide this latency is through pre-fetching. In this thesis, the design of data structures and algorithms for six different object prediction and pre-fetching strategies has been presented. These pre-fetching strategies are based on various combinations of three pre-fetching concepts defined in the thesis: pre-fetching threshold, pre-fetching depth and path-based pre-fetching. All of the algorithms presented are efficient (worst case, quadratic time or better) and relatively simple to implement. A general discussion of the applicability of the various strategies and algorithms was also presented that shows that some of the strategies should be practical for used with existing commodity network hardware operating at speeds of 100mbps and better). The application of pre-fetching in the specific environment of distributed persistent object systems appears to be a novel contribution of the thesis.

6.2. Future Work

A number of open problems were identified in the presentation of this thesis. They all represent areas of potential future work.

The first such problem dealt with the updating of weights in response to an object access. In the algorithms presented, the weight of a new object is initially set to a mid-value (either with respect to other values or the range of the variable used to store the weight if no other weights are yet known). When an object is accessed, its corresponding weight in the reference predictor structure for the accessing object is simply incremented while the weights of all other objects in the structure are decremented. The goal in doing this is to have the weights accurately reflect the relative probability of each object being accessed next. As described in Section 4.5.1, other schemes for updating the object

weights may be better at achieving this goal. In particular it may be useful to select increment and decrement values based on system behaviour. Further, it might be useful to have dynamic values so that when new object reference behaviours are recognized they can be quickly incorporated into the pre-fetching process (e.g. by rapidly increasing the weights of new frequently accessed objects).

It may also prove beneficial to dynamically change other algorithm-specific values. These include the number of <nextOID, weight> pairs, the threshold value (when using strategies that support the concept of pre-fetching threshold), the length of the pre-fetch path (when using strategies supporting path-based pre-fetching) and the pre-fetch depth (again, where appropriate). All of these are, effectively, tunable parameters that can be set dynamically to adjust the pre-fetching behaviour and thereby optimize the system's overall performance.

Dynamically adjusting the number of <nextOID, weight> pairs affects the storage required in all strategies and places an upper bound on the number of objects, which can be pre-fetched. Setting this value on a per object or per object method basis would allow pre-fetching for those objects/methods that may reference a large set of next objects with near equal probability to be effective while not inducing unnecessary storage overhead for those objects/methods for which only a few objects will need to be pre-fetched.

Just as different object methods will tend to reference different sets of objects next so too will different methods tend to reference different *numbers* of methods next. By making the pre-fetching threshold (`ThresholdPctg` in the code) be a method-specific rather than object-specific variable this variance in behaviour could be easily reflected in the pre-fetching actually performed.

It would also be relatively easy to determine how many elements of each stored "object invocation path" actually had an effect on pre-fetching behaviour. Similarly, it would be possible to merge paths with identical (or sufficiently similar) sets of objects to be pre-fetched. Both of these optimizations would decrease the storage requirement for the corresponding reference predictors and the corresponding cost of sending those predictors over the network.

Finally, since different objects may have different *effective* pre-fetch thresholds, it is possible to make pre-fetching threshold (Say N) a per-object parameter. This will save space in the GDO which is an issue because the GDO is distributed and the larger the entries in the GDO are, the greater the cost in distributing them and maintaining consistency between the distributed copies. A reasonable approach might be to start N at a reasonably large value (for new objects). The value of N could then be decreased over time. By monitoring the contents and use of each reference predictor it would be possible to determine that if only a few elements in a given predictor are actually used for pre-fetching. Once this information is known, the value of N for the corresponding object could be decreased and with it the size of the array in the reference predictor could also be decreased. Of course, it may also be that N might have to be dynamically increased in response to changing access patterns. This too is possible.

7. Bibliography

[1] J.H. Ahn and H.J. Kim. SEOF: an Adaptable Object Pre-fetch Policy for Object-oriented Database Systems. Proceeding of the 13th International Conference on Data Engineering, 1997.

[2] T.Alexander and G. Kedem. Distributed Prefetch-buffer/cache Design for High Performance Memory Systems. Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture, 1996.

[3] A. Bahrami. Object Oriented Systems Development. Irwin/McGraw-Hill, 1999.

[4] S.C. Bailine. An Object-oriented Requirements Specification Methods. Communications of the ACM. 1989;32:608-623.

[5] R. Bianchini and T. LeBlanc. A Preliminary Evaluation of Cache-miss-initiated Pre-fetching Techniques in Scalable Multiprocessors. Tech Report 515. University of Rochester, 1994.

[6] G. Booch. Object-oriented Analysis and Design with Applications. Benjamin/Comming Pub. Co, 1994 (2nd edition).

[7] D. Callahan, K. Kennedy and A. Porterfield. Software Pre-fetching. Proceeding of the 4th International Conference on Architectural Support for Programming Language and Operating Systems, 1991.

[8] E.E. Chang and R.H. Katz. Exploiting Inheritance and Structure Semanics for Effective Clustering and Buffering in an Object-oriented DBMS. Proceeding of the ACM SIGMOD International Conference on the Management of Data, 1989.

[9] P.Y. Chang, D.R. Kaeli and Y. Liu. Branch-directed Data Cache Pre-fetching. Proceeding of the 2nd Annual Workshop on Shared-memory Multiprocessor Systems, 1994.

[10] C. Dix. Working with SOAP, the Simple Object Access Protocol. C/C++ Users Journal, 2001;19:22-33.

[11] T.F. Chen and J.L. Baer. Reducing Memory Latency via Non-blocking and Pre-fetching Caches. Proceeding of the 5th International Conference on Architectural Support Programming Language and Operating Systems, 1992.

[12] J.R. Cheng and A.R. Hurson. On the Performance Issures of Object-Based Buffering. Proceeding of the 1st International Conference on Parallel and Distributed Information System, 1991.

[13] O. Dahl and K. Nygaard. Simula-An Algol-based Simulation Language. Communications of the ACM, 1966;9:671-678.

[14] F. Dahlgren, M. Dubois and P. Steustrom. Fixed and Adaptive Sequential Pre-fetching in Shared Memory Multiprocessor. Proceeding of the International Conference on Paralled Processing, 1993.

[15] A. Dearle and D. Hulse. Operating System Support for Persistent Systems: Past, Present and Future. Software Practice and Experience. 2000;30:295-324.

[16] M. Ellis and J. Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley. 1990.

[17] J.W.C. Fu and J.H. Patel. Stride Directed Pre-fetching in Scalar Processors. Proceeding of the 25th Annual International Symposium on Microarchitecture, 1992.

[18] E. Gamma, R. Helm, R. Johnson and J. Vlissides Design Patterns: Elements of Re-usable Object-Oriented Software. Addison-Wesley. 1994.

[19] J.M. Gil, C.Y. Park, C.S. Hwang, D.S. Park, J.G. Shon, Y.S. Jeong. Restoration Scheme of Mobility Databases by Mobility Learning and Prediction in PCS Networks. IEEE Journal on Selected Areas in Communications, 2001;19:1962-1973.

[20] N. Gunton. SOAP:Simplifying Distributed Development. Dr. Dobb's Journal, 2001;26:89-95.

[21] E. Gornish, E. Granston and A. Veidenbaum. Compiler-directed Data Pre-fetching in Multiprocessors with Memory Hierarchies. Proceeding of International Conference on Supercomputing, 1990.

[22] P. Graham and K. Barker. Distributed Object Base Implementation Using a Single Shared Address Space. Proceeding of the Mid-Continent Information Systems Conference, 1993.

[23] M.F. Hornick and S.B. Zdonik. A Shared, Segmented Memory System for an Object-oriented Database. ACM Transcations on Office Information Systems, 1987;5:70-95.

[24] Intasca Distributed Object Database Management System. Technical Report Technical Summary Release 2.0, Itasca Systems Inc, 1991.

[25] D.R. Kaeli, P.G. Emma, J.W. Knight and T.R. Puzak, Contrasting Instruction Fetch Time and Instruction-decode Time Branch Prediction Mechanisms: Achieving Synergy through Their Cooperative Operation. Proceeding of the18th EUOMICRO Symposium on Microprocessing and Microprogramming, 1992.

[26] D. Kemper and A. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases. Proceeding of the 9th International Conference on Data Engineering, 1993.

[27] M.L. Kersten, S. Plomp and C.A Van Den Berg. Object Storage Management in Goblin. In M. Tamer Ozsu, U. Dayal and P.Valduriez, editors, Distributed Object Management. Morgan Kaufmann Publishers, 1994.

[28] A. Ki. Secondary Cache Enhancement Using a Novel Tagged Pre-fetching Method. Microprocessors and Microsystems, 1999;23:245-253.

[29] P. Keleher, A. Cox and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. Proceeding of the 9th Symposium on Computer Architecture, 1992.

[30] M.L. Kersten, S. Plomp and C.A Van Den Berg. Object Storage Management in GOBLIN. In M. Tamer Ozsu, U. Dayal and P.Valduriez. eds: Distributed Object Management. Morgan Kaufmann Publishers, 1994.

[31] N. Knafla. Analysing Object Relationships to Predict Page Access for Prefetching. Proceeding of the 8th International Workshop on Persistent Object Systems: Design, Implementation and Use, 1998.

[32] N. Knafla. A Pre-fetching Technique for Object-Oriented Databases. Technical Report ECS-CSG-28-97. University of Edinburgh, Department of Computer Science, 1997.

[33] R. Kordale and M. Ahamad. Object Caching in a CORBA Compliant System. Computing systems, 1997; 9:377-404.

[34] T. Korson and JD. McGregor. Object-oriented Software Design: a Tutorial. Communications of the ACM, 1990;33:40-60.

[35] A. Kraiss, G. Weikum. Vertical Data Migration in Large Near-Line Document Archives Based on Markov-Chain Predictions. Proceeding of the 23th International Conference on Very Large Database, 1997.

[36] T.M. Kroeger, D.D.E. Long. The Case for Efficient File Access Pattern Modeling. Proceeding of the 7th Workshop on Hot Topics in Operating Systems, 1999.

[37] J. Lee, and A.J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. Computer, 1984;17(1):6-22.

[38] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, AC. Myers, L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. Proceeding of the ACM SIGMOD International Conference on Management of Data, 1996.

[39] Y. Liu, David R. Kaeli. Branch-directed and Stride-based Data Cache Pre-fetching. Proceeding of International Conference on Computer Design, 1996.

[40] J.A. Mathew, P.C.J. Graham, K.E. Barker. Object Directory Design for a Fully Distributed Persistent Object System. Engineering Systems Design and Analysis, 1996; 2:75-87.

[41] S. McFarling and J. Hennessy. Reducing the Cost of Branches. Proceeding of the 13th International Symposium on Computer Architecture, 1986.

[42] B. Meyer. Object-oriented Software Constitution. Prentice Hall, Hemel Hemstead, UK, 1988, 534.

[43] R. Morison and M.P. Atkinson. Persistent Languages and Architectures. Proceeding of the International Workshop on Computer Architectures to Support Security and Persistence of Information, 1990.

[44] Thomas J. Mowbray and Milliam A. RUH. Inside CORBA – Distributed Object Standards and Applications. Addison Wesley Longman, Inc., Sydney, 1997.

[45] T.C. Mowry, M.S. Lam, A. Gupta. Design and Evaluation of a Compiler Algorithm for Pre-fetching. Proceeding of the 5th International Conference on

Architectural Support for Programming Language and Operating Systems, 1992.

[46] E. Mumdo, G. Bernardis. A Novel Demand Prefetching Algorithm Based on Volterra Adaptive Prediction for Virtual Memory Management Systems. Proceeding of the 30th International Conference on System Sciences, 1997.

[47] G. Oliver and A. Laurent. Object Grouping in EOS. In M. Tamer Ozsu, U. Dayal and P.Valduriez, editors, Distributed Object Management. Morgan Kaufmann Publishers, 1994.

[48] M. Palmer and S.B. Zdonik. Fido: A Cache That Learns to Fetch. Proceeding of the 7th International Conference on Very Large Data Bases, 1991.

[49] S. Pan, K. So, J. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. Proceeding of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, 1992.

[50] D. Patterson and J. Hennessey. Computer Architecture – a Quantitative Approach. Morgan Kaufmann(2nd edition), 1997.

[51] R. Peters, P.C.J. Graham, K.E. Barker. A Shared Environment to Support Multiple Advanced Application Systems. Proceeding of the Workshop on Information Technologies and Systems, 1997.

[52] S. Pink, A. Saulsbury, O. Hagsand. OS6-a Distributed Operating System for the Next Generation of Computer Networks. Proceeding of the 4th International Workshop on Object Orientation in Operating Systems, 1995.

[53] S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. Proceeding of the 17th International Symposium on Computer Architecture, 1990.

[54] J.E. Smith. A Study of Branch Prediction Strategies. Proceeding of the 8th Annual International Symposium on Computer Architecture, 1981.

[55] A. Smith. Cache Memories. ACM Computing Surveys, 1982;14:473-530.

[56] M.D. Tarlescu, K.B. Theobald, G.R. Gao. Elastic History Buffer: A Low-cost Method to Improve Branch Prediction Accuracy. Proceeding of the International Conference on Computer Design, 1997.

[57] M.K. Tcheun, H. Yoon, S.R. Maeng. An Adaptive Sequential Pre-fetching Scheme in Shared Memory Multiprocessors. Proceeding of the International Conference on Parallel Processing, 1997.

[58] G.S. Tyson. The Effect of Predicated Execution on Branch Prediction. Proceeding of the 27th International Symposium on Microarchitecture, 1994.

[59] D.L. Wells, J.A. Blakeley: Distribution and Persistence in the Open Object-Oriented Database System. International Workshop on Distributed Object Management, 1992.

[60] S.J. White and D.J. Dewitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. Proceeding of the 18th International Conference on Very Large Data Bases, 1992.

[61] P.R. Wilson and S.V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. Proceeding of the International Workshop on Object Orientation in Operating Systems, 1992.

[62] XML-RPC Home Page: http://www.xml-rpc.com (updated 2002).

[63] B.W. Xu, W.F. Zhang, W. Song, H. Yang, C.H. Chang. Application of Data Mining in Web Pre-fetching. Proceeding of the International Symposium on Multimedia Software Engineering, 2000.

[64] T.Y. Yeh and Y.N. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. Proceeding of the 20th Annual International Symposium on Computer Architecture, 1993.

[65] C. Young, M.D. Smith. Improving the Accuracy of Static Branch Prediction Using Branch Correlation. Proceeding of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.