On the Classification and Evaluation of Prefetching Schemes

Stuart Andrew Blair

Submitted for the degree of
Doctor of Philosophy

Department of Computing Science
University of Glasgow

September 2001
Resubmitted April 2003

Abstract

Despite recognition that prefetching schemes consist of separate prediction and fetching mechanisms [GK94a], previous work in prefetching has failed to evaluate these mechanisms in an independent and portable manner. Consequently, the performance measurements available do not capture the qualities of the individual mechanisms, but rather the prefetching scheme as a single unit. Research and development work based on previous performance results is therefore difficult if not impossible.

The lack of a comprehensive survey of prefetching schemes spanning multiple application areas has bolstered current evaluation practices, since it remains unclear that prediction mechanisms are generic in their applicability to different environments and fetching mechanisms.

This thesis asserts that consideration of prediction mechanisms from different areas exposes universal concepts in prediction and leads to perspectives on evaluation that better inform potential adopters of the technology. Additionally, by examining prediction mechanisms from different domains, this thesis shows that hybrid prediction mechanisms can be devised which incorporate and extend existing work.

This thesis contributes a classification and taxonomy which identifies the fundamental concepts of prediction and fetching mechanisms. This aids future research by identifying opportunities for development in prefetching. The thesis also provides researchers and software engineers with an approach to evaluation which captures the qualities of prediction mechanisms in a way which is portable to other contexts.

Originality of Composition

I declare that the work presented in this thesis embodies the results of my own special work, that it has been composed by myself and that it does not include work forming part of a thesis presented successfully for a degree in this or another University.

Stuart A Blair

This work is dedicated to the people who have supported me throughout my PhD, and without whom, the experience would have been far emptier, lonlier and more daunting. In particular I would like to thank the following people.

- My partner, Pamela for her love, comfort and affection.

- My parents, for their emotional support and unstinting belief in my abilities.

- My supervisor Quintin Cutts, for his level-headed, clarifying influence.

This work was funded under an EPSRC studentship. An additional year of funding was provided by the SUN Microsystems grant held by the HOPS group, for which I am extremely grateful.

Contents

List of Figures

Chapter 1

Introduction

The increasing performance disparity between magnetic disk storage devices, main memory, and microprocessors imposes a major bottleneck in the execution of large software applications. Since the 1970s, the growth in microprocessor speed has followed Moore's law by doubling every 18 months to 2 years [sia99] (66% annually). By comparison, improvements in the performance of memory have progressed at the rate of 8-10% annually, while the performance of magnetic disks have been limited as a result of their reliance upon moving parts.

Prefetching is an optimisation technique aimed at reducing the impact of this performance disparity on program execution. The technique involves predicting the data which will be required in the executing application's near future and arranging for it to be brought in from secondary storage to memory before it is required by the application. The fetching of the data takes place in parallel with the ongoing execution, and so the cost is hidden.

Prefetching schemes have been designed for a number of application areas ranging from web servers [Bes95], file systems [GA94, KE90, MJLF84], and Object Oriented Database Management Systems (OODBMS) [PZ91, Kna97a, GK94b, GK94a, CKV93, CK89] to scientific applications [Tri76, KKP94, MLG92]. Despite this, no survey of prefetching schemes exists to span

the many application areas in which prefetching has been applied. This thesis provides a system of classification for prefetching by examining the mechanism used to predict data accesses and the mechanism to make predicted data resident. The thesis then proceeds to a comprehensive cross-application survey based on this classification. The classification considers the use of prediction mechanisms in different application areas in a manner which is orthogonal to the fetching mechanism used. Additionally, the classification examines the many environmental dependencies affecting the performance of a prefetching scheme.

The survey also reveals that performance evaluation of prefetching schemes in the literature has been carried out in a manner which abstracts over many variable dependencies including the application style, operating system and hardware. As a result, the performance evaluation results cannot be used to make meaningful comparisons on the efficacy of prefetching schemes or the prediction mechanisms which guide them. Comparative evaluations based on published performance results would therefore fail to compare like systems with like. Accordingly, such evaluations fail to inform potential adopters of the technology whether or not it is likely to be of benefit on their particular combination of application, operating system, hardware etc. This presents a major obstacle to research which attempts to build on the accomplishments of existing mechanisms, and provides a plausible explanation as to why many of the prediction mechanisms have been developed in isolation from their predecessors.

This thesis submits that generic evaluation of prediction mechanisms using a universal metric is not possible. Instead, the thesis embraces this and presents an alternative approach to the evaluation of prefetching schemes and suggests ways in which portable and meaningful performance measurements can be obtained. This is accomplished by identifying fundamental requirements of successful prediction and developing targeted micro-benchmarks which expose the performance of particular classes of prediction mechanism with respect to these requirements.

The survey's classification and separate treatment of prediction and fetching mechanisms encourage the development of hybrid prefetching schemes which seek to incorporate the advantages of prediction and fetching mechanisms from different application areas. This thesis presents a hybrid prediction mechanism called Sympa which is designed to exploit the environment of object oriented orthogonally persistent systems. The novel aspect of Sympa is that it combines the advantages of several prediction mechanisms spanning the spectrum of the classification presented by the thesis but itself is applied in a new context. In addition, it has been developed to run upon an existing fetching mechanism, thereby demonstrating the orthogonality of fetching and prediction mechanisms.

The prediction mechanism is then evaluated using the approach to evaluation proposed by the thesis.

1.1 Thesis Statement

This thesis asserts that consideration of prediction mechanisms from different areas exposes universal concepts in prediction and leads to perspectives on evaluation that better inform potential adopters of the technology. Additionally, by examining prediction mechanisms from different domains, this thesis shows that hybrid prediction mechanisms can be devised which incorporate and extend existing work.

1.2 Thesis Contribution

The contributions of this thesis lie in three areas. Firstly, a comprehensive survey of prefetching schemes spanning multiple application areas is presented leading to a classification scheme for predictors and fetchers. This survey presents a taxonomy of the prediction and fetching mechanisms employed by over 20 prefetching schemes found in the literature.

From the survey comes the motivation for the second contribution: a critique of evaluation methods used in the literature to date, and the proposal of a more effective approach which better captures the qualities of the prediction mechanisms.

The third contribution is Sympa: a hybrid prediction mechanism for persistent OO languages. Sympa and its evaluation demonstrate the worth of the first two contributions. The hybrid mechanism is derived from elements presented in the cross-application survey, and the evaluation applies the method presented in the second contribution.

It is intended that this thesis will guide future research and development in prefetching by providing researchers and software engineers with:

- a consolidated treatment of previous work in prefetching which exposes the fundamental concepts common to all prediction mechanisms. The classification will aid research by highlighting new opportunities for prefetching schemes. The taxonomy will be useful for those looking to deploy existing work.

- an appreciation of the difficulties in producing performance results of a prediction mechanism which are portable to other contexts.

- an approach to evaluation which better captures the qualities specific to a prediction mechanism rather than its use in the context of a particular combination of machine, OS, and application. This will enable researchers seeking to create future prediction mechanisms to recognise the qualities of particular prediction mechanisms and make informed choices on whether to adopt them in their own context. This work will also enable software engineers to make judgements on the suitability of a prediction mechanism to their application by comparing micro-benchmark code to that of their application.

1.3 Terminology and Conventions

In this work, the term "application" is considered to relate to the use of a computer system to accomplish a goal. To achieve this goal, application programs execute on the computer system. In this context, an application may be taken to mean a general area of use such as CAD, or web browsing. The "application program" is taken to mean a particular computer program which is applied to the application.

Text in italics deals with previously undiscussed terms which are to be discussed within the current passage of text. Program fragments are formatted in courier and written in Java.

Other terms are introduced as and when they are required by the chapters which follow.

1.4 Contents and Layout

The rest of the thesis progresses through the following chapters. Chapter 2 introduces the concept of latency and how it arises. It also introduces the various means employed to lessen the impact of latency: caching, clustering, and prefetching. The requirements of a prefetching scheme are then examined.

Chapter 3 presents a survey of the fetching mechanisms used by prefetching schemes to make data resident ahead of its use by the application. The survey presents a taxonomy of fetching mechanisms along the dimensions of run-time cost, unit of transfer, and intelligence. In this way the survey highlights the advantages and disadvantages along each dimension.

Chapter 4 presents a survey of the prediction mechanisms used by prefetching schemes to predict which data the application will require in the future.

Chapter 5 discusses the goal of prefetching in relation to the methods used to evaluate it. This discussion highlights the weaknesses in purely time-based measurements and proceeds to explain

the many factors affecting prefetching performance. Alternative evaluation methods are discussed before an approach to evaluation is proposed to enable the separate and portable evaluation of prediction and fetching mechanisms.

Chapter 6 demonstrates the utility of the evaluation framework proposed in chapter 5 and verifies the results against a Java implementation of a First Order Markov predictor running over a OO7 benchmark application.

Chapter 7 presents the concepts involved in Sympa, a prediction mechanism for OO persistent languages. It introduces the reference shape and explains its place in predicting the referencing behaviours of applications. The chapter also discusses how reference shapes can be applied to object graphs to predict page accesses.

Chapter 8 uses the approach to evaluation proposed in chapter 5 to create bespoke benchmarks which demonstrate the strengths and weaknesses of Sympa under different types of application. Comparison of those application types is made with both the OO7 benchmark and the GAP GIS application. The evaluation experiments are described in detail and the results are presented. The chapter concludes by reflecting upon the effectiveness of Sympa in the different situations posed in the evaluation.

Finally, chapter 9 presents a summary of the discoveries and achievements of this work and makes suggestions for future work in this area.

Chapter 2

Latency and Prefetching

This chapter introduces the concept of latency and presents quantitative evidence of the prevailing hardware trends which cause it. The memory hierarchy of computer systems is presented to show the points where latency manifests itself. Latency optimisations are then introduced which address the negative impact of data retrieval over the memory hierarchy. The relationships between these optimisations are discussed and, in view of hardware trends, prefetching is examined in detail. A brief overview of the application areas for prefetching is also presented before a summary of the material is presented.

2.1 Latency

In general terms, latency can be defined as the elapsed time between a stimulus and its response. In the context of computer systems, latency can be defined as the time between the request for a unit of data and the point where it is made available to the component of the system which requested it. More precisely, latency is the time delay experienced by computer systems while data is retrieved from *data staging areas*. So, the request for data is the stimulus and the response corresponds to the receipt of the requested data.

The effects of latency manifest themselves when data is transferred between the components of a computer system. For ease of exposition, this section introduces the term *data staging area* to refer to any component which the computer uses to store and retrieve data. This term encompasses devices ranging from server-mounted magnetic disks to microprocessor registers.

A model of the latency costs in a typical network connecting two computers is given in [HP96]. This model is generalised below to show the composition of latency costs for other data staging areas. The total latency L_{total}, between the request and receipt of data in full is defined using the following terms.

- O_{sender}: the overhead time taken for the microprocessor to issue the request for the data to the appropriate data staging area. This period represents the relatively small time that the microprocessor spends requesting data.

- $O_{receiver}$: the overhead time taken for the microprocessor to transform the data into a form used by the on-going execution or to adjust any housekeeping data structures to reflect the new state of data storage areas. In general, the receiver overhead is larger than the sender overhead.

- T_{flight}: the time for the first unit of data to arrive at the destination for data: either microprocessor or data staging area. This measure does not account for the time taken to retrieve the data from a data staging area in its entirety.

- $T_{transmission}$: the elapsed time between the first and last units of the requested data reaching their destination: either microprocessor or data staging area.

Given these definitions, the total latency may be expressed as:

$$L_{total} = O_{sender} + T_{flight} + T_{transmission} + O_{receiver}$$

2.1.1 Memory Hierarchy and Degree of Latency

In an ideal world, computers would suffer no latency when reading data to process. Under this impossible scenario, computers would be limited only by the speed of the microprocessor. Unfortunately, all data retrieval and storage operations incur some time penalty, whether induced by flipping the state of logic gates, or by mechanically moving arms and platters in magnetic disks.

Computer systems process data from a layered hierarchy of data staging areas. The latencies, bandwidth and cost per-byte stored vary greatly between the data staging areas used to implement the layers of the memory hierarchy. Figure 2.1 specifies the typical latencies and storage capacities of data staging areas used in current memory hierarchies.

Since it is prohibitively expensive to have all data stored in staging areas with low latencies, the memory hierarchy is structured in such a way as to have a small, low latency layer backed by successively larger layers with higher latency.

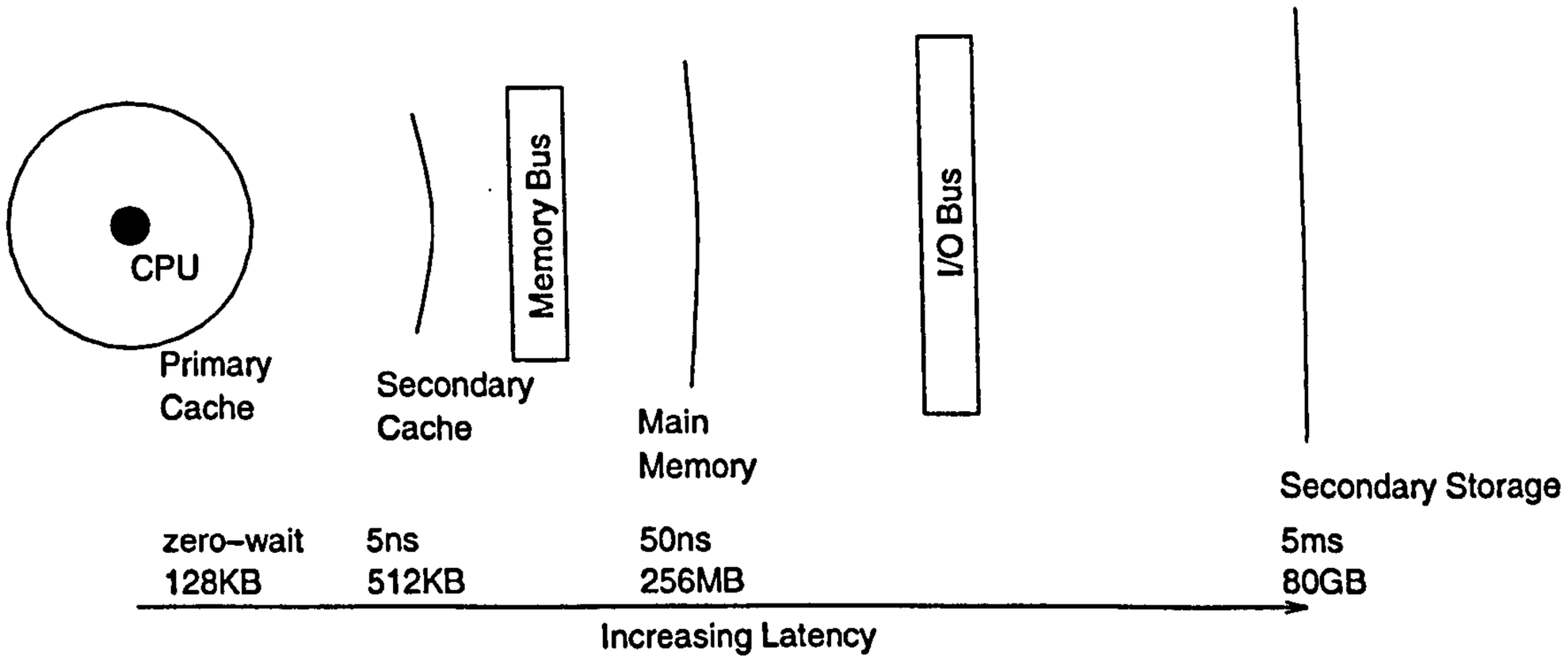

Figure 2.1: Latency expressed as distance between the microprocessor and data staging areas.

2.1.1 Memory Hierarchy and Degree of Latency

In an ideal world, computers would suffer no latency when reading data to process. Under this impossible scenario, computers would be limited only by the speed of the microprocessor. Unfortunately, all data retrieval and storage operations incur some time penalty, whether induced by flipping the state of logic gates, or by mechanically moving arms and platters in magnetic disks.

Computer systems process data from a layered hierarchy of data staging areas. The latencies, bandwidth and cost per-byte stored vary greatly between the data staging areas used to implement the layers of the memory hierarchy. Figure 2.1 specifies the typical latencies and storage capacities of data staging areas used in current memory hierarchies.

Since it is prohibitively expensive to have all data stored in staging areas with low latencies, the memory hierarchy is structured in such a way as to have a small, low latency layer backed by successively larger layers with higher latency.

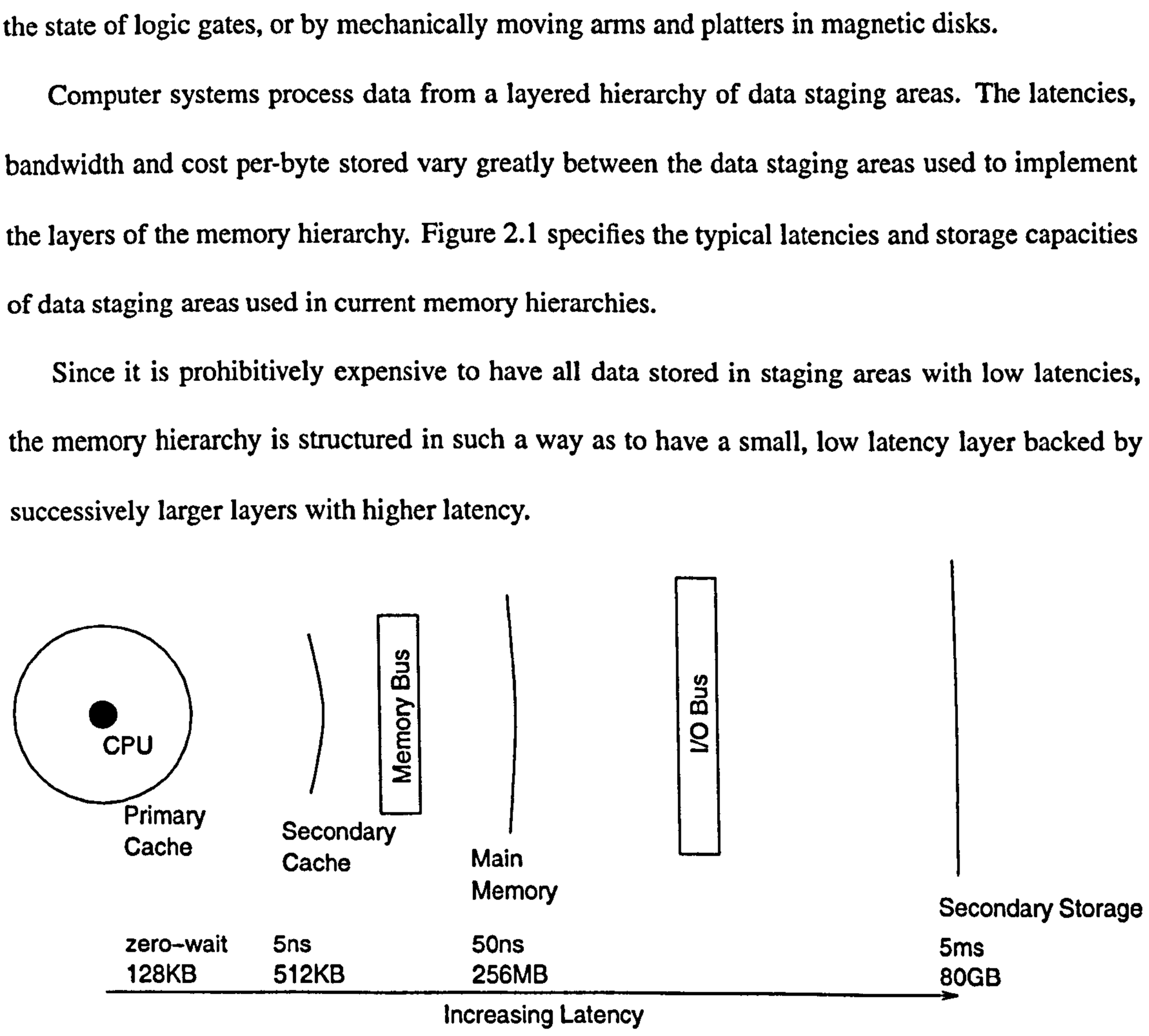

Figure 2.1: Latency expressed as distance between the microprocessor and data staging areas.

Data Locality

During program execution, data is moved closer[1] to the microprocessor through the layers of the memory hierarchy. This process incurs time penalties on program execution as a result of layer latencies.

Clearly, the memory hierarchy cannot provide uniform access times for data held at different levels. Ideally each data access would incur only the latency penalty of the fastest data staging area. Such areas are limited in size. However, the memory hierarchy exploits the heuristic that when a data item is used, it will be re-used in the near future. This is more formally referred to as the principle of *Data locality* [PH94]. The principle states that applications access a relatively small portion of their address space at any instant of time. There are two types of data locality:

- *Temporal locality*: If an item is referenced, it will tend to be referenced again soon.
- *Spatial locality*: If an item is referenced, items whose addresses are close by will tend to be referenced soon.

While data locality has the potential to produce lower miss rates in the faster layers than might otherwise have been expected, the cost of the initial miss (*cache miss*) still imposes a significant penalty upon program execution.

Latency Barriers

Figure 2.1 shows that the difference in latencies between successive layers in the memory hierarchy is not constant. The largest differences between neighbouring layers can be found between the layers separated by the memory bus and the layers separated by the I/O bus.

[1] In the sense of latency being expressed as distance from the microprocessor

- At the memory bus between cache and main memory the difference in latencies is typically an order of magnitude.

- At the I/O bus between main memory and disk the difference in latencies may be up to 5 orders of magnitude.

These constitute *latency barriers* which attract attention from system architects and programmers interested in optimising performance.

The latency barriers arise as a result of the different materials, construction methods, and organisation in the technologies used to implement each layer of the memory hierarchy. Primary and secondary cache memory are both implemented using static random access memory (SRAM), main memory uses dynamic random access memory (DRAM), while secondary storage is implemented using magnetic disks.

Primary cache is located on the microprocessor in such a way as to have a zero wait-state (delay) interface to the microprocessor. SRAM is based on the use of flip-flops to maintain a stable state and offers extremely low access speeds. Each bit in memory is stored using an arrangement of between 4 and 6 precisely located transistors which unfortunately makes it prohibitively expensive to have large amounts of SRAM in the system.

DRAM uses arrays of cells with support logic to perform the reading and writing in addition to capacitor circuitry to maintain the state of the cells. DRAM is manufactured using a silicon substrate etched with a simple repeating pattern comprising transistors and capacitors to represent each bit. This requires a less complex manufacturing process than that used in SRAM and results in a comparatively lower cost. However, the need for the capacitor to refresh the state of the memory cell impedes access causing DRAM to be dramatically slower than SRAM.

Secondary storage commonly relies upon the use of magnetic disks. These mechanical devices have a system of glass platters with a magnetic thin-film medium which stores data in magnetic

patterns. There are typically 3 platters mounted on a central spindle. A system of heads is moved radially on an arm to cover all parts of the disk for reading and writing data.

2.1.2 Hardware Trends

The scenario depicted in figure 2.1 shows the current performance disparities between the different layers of the memory hierarchy. Since the layers of the hierarchy rely upon different technologies, the rates of performance improvement may change the relative size of the latency barriers. Analysis of the trends in performance improvement are presented here for magnetic disks, memory, and microprocessors.

Improvement in Magnetic Disk Performance

Although there have been dramatic improvements in the rate of data transfer or *bandwidth* in magnetic disks, improvements in the time taken to get a random block of data from a disk have been less remarkable.

The improvement in magnetic disk bandwidth is due to the increased density with which information can be recorded (*areal density*) and the increased rotational velocity of the drives. Increased areal density brings greater storage per unit area. Increased rotational velocity brings faster coverage of the disk area. These two factors have resulted in more information being read from the disk in any given instant.

In order to generate higher areal densities, smaller magnetic fields must be generated to record the bits closer together. This has been possible due to manufacturing advances in disk head technology. The improvement in areal density is charted in figure 2.2.

The limit to the continuing growth in areal density is determined by the size of the magnetic fields generated by the heads. As the field becomes smaller in order to affect a smaller area on

the media surface, the thermal energy of the environment will have an increasingly destabilising influence on the state of the field storing the bit state.

In order for a disk block to be read from or written to, the disk heads have to be moved radially to the track containing the block. This is the *seek time* of the disk. Once the heads have moved, the controller must wait for sector containing the block to pass by the heads. The time is represented by the *average rotational latency* and can be computed as half the time taken for one complete rotation of the disk. The average time taken to read a random block of data (*mechanical latency*) therefore consists of the seek time and rotational latency. Although the increases in rotational velocity have reduced the rotational latency, the improvements in seek time have been modest as a result of the reliance upon moving the mechanical arms with sufficient speed and accuracy.

Accordingly, as advances are made in other aspects of magnetic disk technology, this reliance upon arm movement accounts for an increasingly large proportion of the cost in random block accesses.

Since many applications read contiguous blocks of data, it has become a common strategy to have an on-disk cache. This cache houses a copy of the most recently accessed sector or cylinder making subsequent accesses to that sector or cylinder very inexpensive by comparison to another disk access. Unfortunately, these caches are limited in size (typically less that 4MB) and are of little use if the application does not exhibit a high degree of data locality.

Improvement in Memory Performance

Both SRAM and DRAM memory are improving at the same rate of 8-10% a year. Since similar materials are used in both types of memory, advances in manufacturing processes have affected both equally.

Innovations in the organisation of memory devices are enabling ever-higher bandwidths and

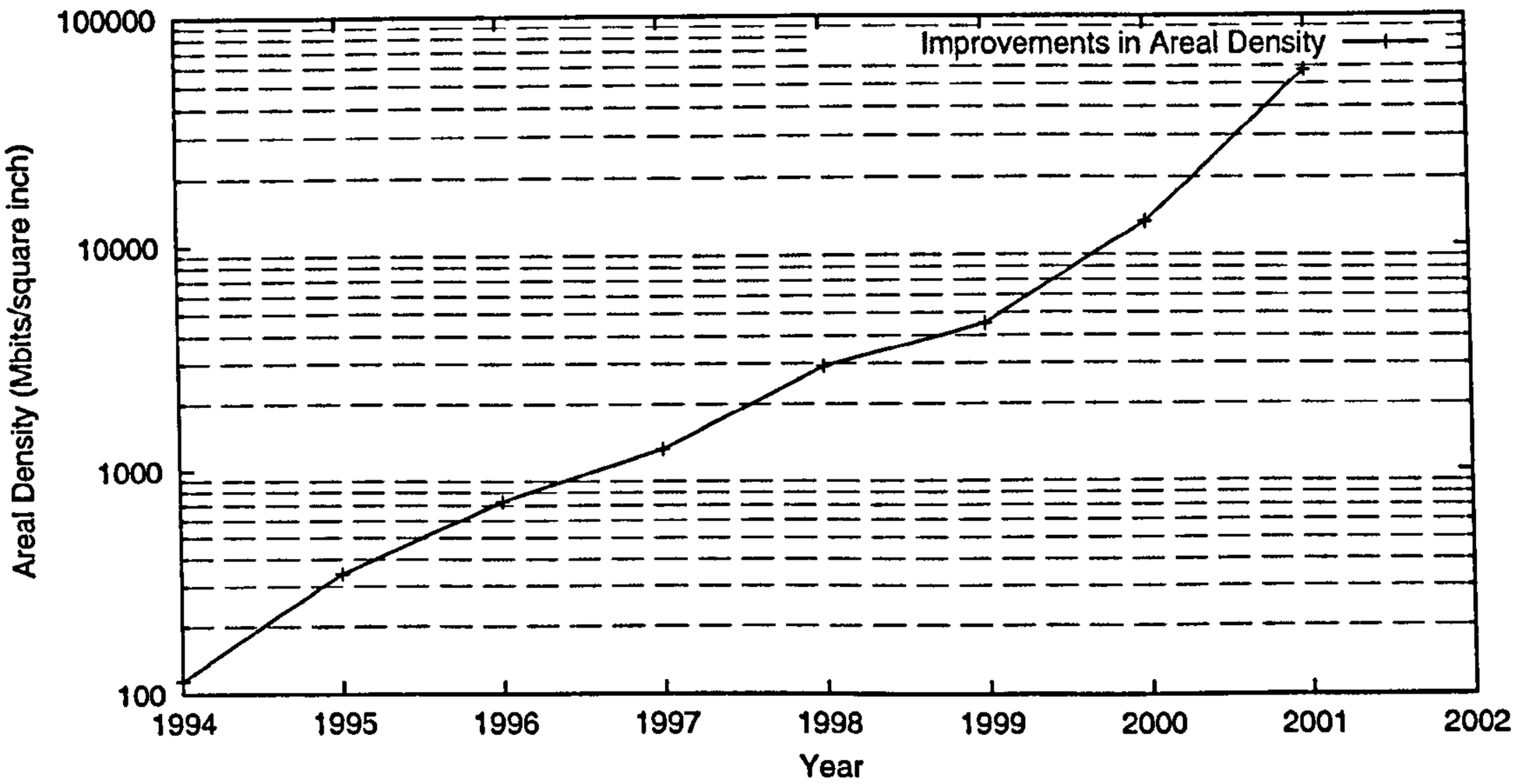

Figure 2.2: Improvement in areal density of magnetic disks.

lower latencies [sia99].

Improvement in Microprocessor Performance

In terms of internal clock speed, microprocessors are following the upper bound of Moore's law: doubling in performance every 18 months. This is equivalent to an annual increase of 66%. These improvements are due to advances in manufacturing technologies which allow more components to be etched onto a chip and for them to be placed closer together. This lessens the power consumption of the chip and allows it to be driven at higher frequencies.

Additionally, improvements in microprocessor design such as super-scalar architectures, SIMD, and speculative parallel execution with large primary caches means that more work per clock cycle can be done by the microprocessor.

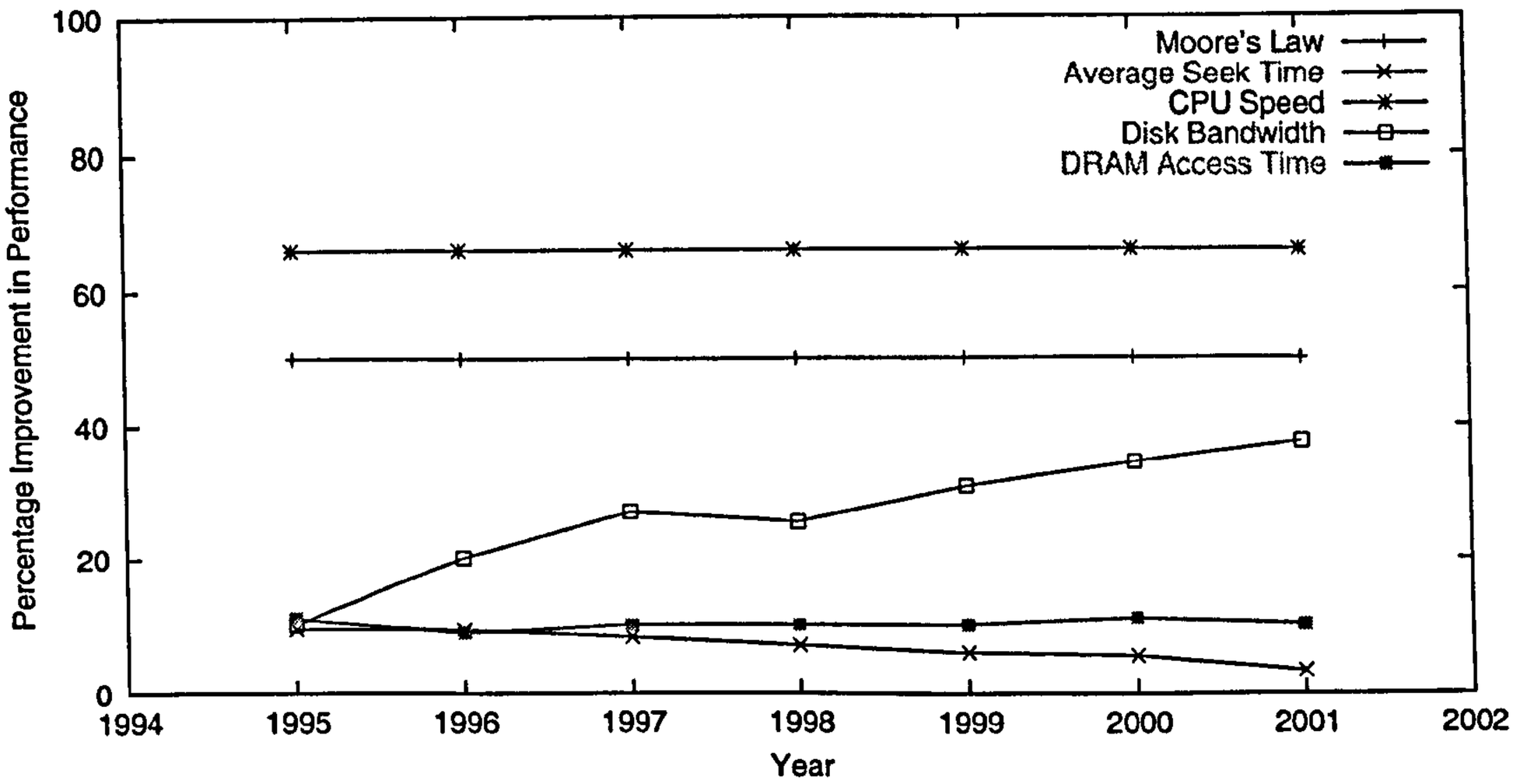

Figure 2.3: Percentage improvement in performance of microprocessor, memory, disk media transfer rate and disk seek time.

Overall Trends and Future

Analysis of the trends in microprocessor, memory, and magnetic disk, shows the different rates of improvement (figure 2.3). These trends are responsible for the increasingly detrimental effect of latency in data staging areas.

Given the growth rate of microprocessors relative to that of memory and magnetic disk, it is unlikely that microprocessor throughput will become a bottleneck for most applications.

Considering the rate of improvement in memory latency compared to that of magnetic disk, it is likely that applications which rely upon large quantities of persistent data will remain bound by the performance limitations of magnetic disk. Scientific applications performing matrix calculations upon in-core data will continue to be bound by memory latency.

Although developments in alternative devices such as Magnetic Dynamic Random Access Memory (MDRAM) may eventually replace hard disks, the performance gaps will remain. Ultimately, as

long as the memory hierarchies exist, latencies will act as a barrier to program execution.

2.2 Latency Optimisations

A number of software-based optimisations have been proposed [AK97, BKW94, BS96, CFKL95, CFKL96, CK89, Cha89, CH91, CKV93, FCL93, GKKM92, GK94a, GK94b, GLC+92, GA94, GAN93, HMMS98, KGM91, KKP94, Kna97c, Kna97a, KE90, Lam88, Li92, LM96, MK94, MJLF84, MLG92, MDK96, PZ91, PGG+95, RPASA97, Smi78, TPG97, Tri76, TN92] which attempt to improve the performance of applications as data is transferred across the layers of the memory hierarchy. These optimisations can be categorised as either *latency reduction* or *latency tolerance* optimisations.

Latency reduction optimisations attempt to reduce the number of data access operations which directly involve higher latency layers. By contrast, latency tolerance does not address the number of operations involving lower layers, but instead tries to lessen their impact on run-time performance. This is accomplished by the careful scheduling of data access operations in such a way as to allow the maximum microprocessor throughput.

Latency incurred by operations which access data staging areas is categorised into *read latency* for those operations which retrieve data, and *write latency* for those which store data. Although there are widely accepted techniques to reduce the impact of write latency on total execution time, reducing the impact of read latency requires advanced knowledge of the current program's future behaviour.

2.2.1 Latency Reduction

The two most commonly used latency reduction optimisations are those of caching and clustering. Both of these exploit data locality (section 2.1.1).

Caching exploits referential locality. When data is accessed and brought into a layer of the memory hierarchy which employs caching, a *cache manager* uses a policy to determine when the data item will be replaced with some other piece of data which has been accessed. The policy is usually based on the frequency of the data access.

Clustering exploits spatial locality. When data is accessed from a layer in the memory hierarchy which employs clustering, it is brought brought to a higher layer along with other data located on the same chunk of data (clustering unit). The clustering units contain items of data which have been co-located on the basis of their relation to each other. There are a number of heuristics to define this relation.

In both caching and clustering, the number of expensive data access operations can be significantly reduced, although not eliminated since the initial transfer from lower to higher layer must still take place.

2.2.2 Latency Tolerance

The key to tolerating latency is to separate the *request* and *use* of data from a lower layer in a way which exposes the inherent parallelism in an executing application. With this approach, the microprocessor spends less time waiting on data by fetching it and finding useful compute-bound work to do while waiting on the data arriving in the desired layer.

Multi-threading

Multi-threading relies upon a pool of concurrently executing threads to exploit useful parallelism. When a thread requests data which will result in access of a lower layer, the thread is blocked and other threads in the pool are executed for the duration of the data movement between layers.

Prefetching

Prefetching relies upon knowledge of the application's future data access behaviour to exploit its inherent parallelism. The separation between the request and use of data is enabled by a special non-blocking operation which fetches the data while allowing the application to continue processing up to the point where the data is used.

2.2.3 Relationships Between Latency Optimisations

Prefetching, caching, and clustering are all similar in that they use policies which predict future data access behaviour in order to lessen the impact of latency upon program execution.

Caching can be seen as dynamic re-clustering where the the cache acts as the clustering unit.

2.3 Focus on Prefetching

Given the performance trends presented in section 2.1.2, as the microprocessor performance and device bandwidth continue to grow, the percentage of program execution time spent suffering cache misses will become larger. This indicates that measures such as prefetching which attempt to hide the cost of cache misses will become increasingly important.

This section presents prefetching in terms of its basic requirements and mechanisms.

2.3.1 An Example of Prefetching

Program 2.1 A simple code fragment.

```
.
.
foo := foo ++;
bar := foo + bar;
.
.
```

Figure 2.4 illustrates the execution of the code in the code fragment of program 2.1 within both prefetching and non-prefetching environments. Periods of microprocessor execution are marked by the lightly shaded areas, whereas the dark areas correspond to the time when the microprocessor is idle, waiting for data from a data storage device.

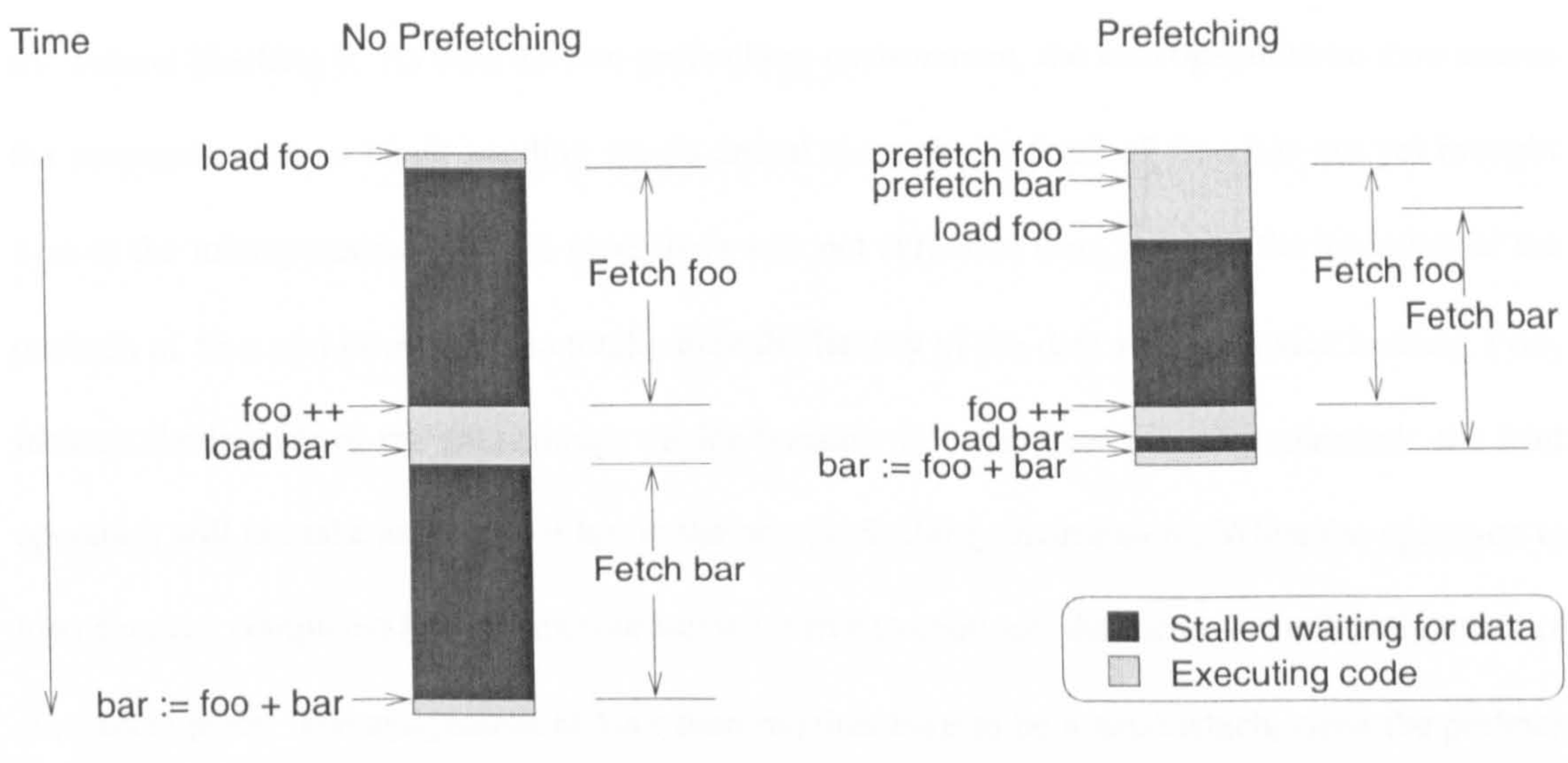

Figure 2.4: How prefetching improves performance by overlapping data retrieval with program execution.

Under the non-prefetching environment in figure 2.4, the increment of the variable `foo` requires knowledge of the present value in `foo` which must be loaded from its data storage device. Since the increment depends upon the value in `foo`, the increment operation cannot proceed until the present value in `foo` is retrieved from the data storage device that houses it. The microprocessor is idle during this period. When data becomes available to the microprocessor, the increment operation may proceed, with the result being stored in the microprocessors local memory. The assignment of the variable `bar` is dependent on the addition (and hence the values) of both `foo` and `bar`. The previous assignment left the value of `foo` in the microprocessor's local memory, however, obtaining the value of `bar` requires a fetch from a data storage device which leaves the microprocessor idle

before the assignment can be made. As can be seen from figure, the total execution time is dominated by periods where the microprocessor is blocked pending data from a data storage device.

Under the prefetching environment in figure 2.4, the data required (`foo` and `bar`) are predicted and the prefetching mechanism initiates the movement of these data items closer to the microprocessor without blocking it. As with the non-prefetching environment, the load operation on `foo` causes the microprocessor to block pending `foo`'s arrival since the prefetch of `foo` has not yet brought `foo` to the microprocessor. In this case, there was not sufficient time between the initiation of the prefetch of `foo` and its retrieval to totally hide the latency of the data storage device housing `foo`. Instead, the latency of the data storage device housing `foo` has been partially tolerated: the load operation will not take as long as it has in the non-prefetching environment. When the operation to load `foo` has completed and the microprocessor is free to continue, the increment and assignment of `foo` takes place. The assignment of `bar` then requires `bar` to be loaded which, since the prefetch operation has almost completed moving `bar` towards the microprocessor, will occur without the microprocessor being blocked for an extended period. The addition and assignment can then take place as in the non-prefetching case.

It should be noted that under the assumptions of a pure prefetching architecture, the prefetch operations for `foo` and `bar` are executed in parallel with both the application and with each other. In this way, two separate requests are made (in parallel) of the data storage device housing `foo` and `bar`. In a prefetching system with clustering, or in the case where by fortune `foo` and `bar` are co-located on the same unit of transfer between data storage devices, the fetch of `foo` and `bar` are retrieved with a single I/O operation.

2.3.2 Requirements of Prefetching

Prefetching requires a mechanism to identify (*predict*) data items which are likely to be used and are therefore candidates for prefetching.

Prefetching also requires a mechanism to move the predicted data items closer to the microprocessor ahead of their reference by the on-going execution.

These mechanisms provide useful dimensions along which prefetching techniques may be classified.

Prediction Primitives

Prediction techniques can be classified as belonging to either (or a combination of) the following two methods.

1. **Data access as experienced by a data manager of a data staging area** (eg. device driver, or object cache manager). In this case, the process of predicting which data items should be prefetched is guided by information available to the data manager. This information may include

   - the address of the data currently being processed
   - the type of the data currently being processed
   - the pattern of data access formed by recent executions

2. **Data accesses as issued by the executing application.** Analysis of the application's possible behaviour is used to predict the data items which are likely to be accessed during execution. Since the analysis is not concerned with the data access patterns experienced by any data manager, the prediction is valid only for the object accesses made by the executing application.

Units of Prediction

Predicted data access patterns may be expressed in terms of a sequence of data units. Popular choices [GK94a] for these *units of prediction* vary depending upon the application area. In the case of prefetching schemes for persistent object systems, objects (typically tens or hundreds of bytes) or disk pages (typically 8K) are popular choices for the unit of prediction. In the case of lower-level schemes in use in operating systems, disk blocks [GAN93] or even individual bytes of data [Mow94] are possible representing the values of scalar variables.

Effectiveness

Prefetching data is only *possible* in the presence of knowledge of where to obtain the data (its address) prior to its reference. For example, if the address is dependent on data that is only available immediately before the reference, it may not be possible to compute the address far enough in advance to initiate or complete a prefetch of the data.

Pure prefetching as it has been introduced so far may not be *effective* in reducing execution times under a number of situations. The following conditions are examples of some problems involving effectiveness which will be dealt with in full later. For now, they are presented to illustrate the problems of effectiveness in prefetching. Effectiveness of a pure prefetching strategy may diminish:

- if prefetches are issued for items of data which are already present in the cache, or for which prefetch operations have already been issued. In these cases, the microprocessor overhead incurred by issuing the prefetches is wasted.

- if, due to the timing of a prefetch operation, the prefetched data is not found in the cache when the executing application needs it. This may happen when the prefetch is initiated too late to cope with the latency of the data staging area housing the data.

- if the mechanism that predicts the required data carries an overhead greater than the benefit obtained by having the data close to the microprocessor.

2.4 Related Work in Prefetching

Prefetching has been used to increase execution speed in a number of areas ranging from operating and file systems [GA94, PGG+95, BKW94, CFKL96, TPG97, GAN93] and scientific applications [MLG92, KKP94, HMMS95] to object-oriented databases [GK94b, Kna97a, PZ91, AK97, CKV93]. The methods of predicting data access patterns and making data resident vary due to the type of constraints imposed by the environment in which prefetching takes place. Among the constraints which characterise these environments are

- the degree of latency suffered by fetching non-resident data
- the ratio of data fetch speed to data processing speed
- the predictability of data access patterns
- the (hardware and software) support available

For example, in the case of database prefetching schemes, latency is incurred when memory misses cause data to be transferred from disk or from the network. However, in the case of prefetching in applications with working loads which fit in core memory, latency is suffered when data misses in the on-board cache cause data to be transfered from core memory. In the aforementioned cases, the degree of latency suffered in relation to the rate at which data may be processed varies greatly. Compared to the latter, the former will find the microprocessor sitting idle for longer in the case of a miss. As a result, a predictor which utilises the idle microprocessor might be used in the former case. In the latter case, the overhead of such a predictor would be prohibitively expensive to implement.

2.5 Summary

This chapter introduced the concept of latency in general terms as the time between a request for data and the satisfaction of that request. Latency barriers were presented as layers of the memory hierarchy separated in by a gulf in access speeds.

In order to show how the importance of latency barriers may change, an analysis of hardware trends for magnetic disks, memory, and microprocessors was made which charted the annual percentage improvement in each technology. The continuing divergence in performance between these devices is causing increasingly large latency barriers from memory cache to memory and from memory to magnetic disk. In conclusion, with the rise in microprocessor capacity as well as memory and magnetic disk bandwidths, cache misses will become responsible for an increasing percentage of total execution time.

Ultimately, regardless of the supporting technologies, as long as there is a memory hierarchy separated by different access speeds, there will be a need for latency optimisations to improve performance.

Latency optimisations seek to either tolerate or reduce the detrimental impact of latency on execution time. The relationships between these optimisations were then discussed. In light of hardware trends, prefetching was then discussed in more detail. The discussion included the requirements of prefetching including prediction and fetching mechanisms.

Chapter 3

Making Predicted Data Resident

As described in chapter 2, prefetching requires fetching mechanisms to support the retrieval of predicted data in advance of its reference in the application. The key requirement of such mechanisms is that they should allow the application to continue to execute unimpeded while the predicted data is fetched. This requirement is necessary since prefetching will not be effective in reducing an application's total execution time if the latency penalty incurred while accessing predicted data cannot be hidden.

The fetching mechanisms found in the literature tend to use one of two methods:

- *Explicit fetching* forces the immediate retrieval of a data item in parallel with the application's execution.
- *Indirect fetching* relies upon a manager which receives hints on future accesses and, based on some strategy, decides whether to fetch the data. Explicit fetches are required in order to support this method of prefetching.

The choice of method for a given prefetching scheme depends upon both the operating system and hardware support available to the designers of the prefetching scheme and the degree of latency that

occurs as a result of data fetches.

A number of factors (discussed in the course of this chapter) expose explicit fetching mechanisms as being over-simplistic. In many applications, the naive assumptions made by the explicit mechanisms can cause performance degradation compared to a non-prefetching system.

In addition to the fetching mechanism itself, policies for determining both when prefetch requests are made as well as the granularity of data to be prefetched are important. Beginning with the operating system and hardware support required for prefetching, this chapter discusses the mechanisms present in the literature for making predicted data resident.

3.1 Support for Fetching Mechanisms

This section summarises some of the possible approaches to achieving parallelism between computation and I/O in modern computer systems.

3.1.1 Hardware and Operating System Support

Many fetching mechanisms [CFKL96, HMMS95, KKP94, LM96, MLG92, Mow94, MDK96, PGG+95] rely upon the provision of a prefetch instruction or system call which fetches data into the cache[1]. This prefetch instruction has the special property that it is non-blocking: it issues the requests for data, but the microprocessor may continue processing subsequent instructions without waiting for the requested data to become resident in the cache.

To prevent the corruption of the semantics of applications using non-blocking prefetches, a second property must hold. This property ensures that when a value is accessed by a prefetch instruction, the most recently written value is returned, even if that value was written after the prefetch was

[1] In this chapter and those which follow, the term cache will be used in a general sense to describe an area in some level of the memory hierarchy which stores the data prefetched from a slower layer in the memory hierarchy.

issued. A prefetch which brings a data value to a cache and guarantees that (upon a subsequent load operation) the most recent value of the data item is obtained is called *non-binding* [HP96, Mow94] since the data value is not bound to a local copy. The issue here is one of *coherence* between the levels of the memory hierarchy.

Modern microprocessors and operating systems support non-blocking, non-binding prefetch operations at a number of levels in the memory hierarchy. The SPARC v9 [Sun97] instruction set includes a prefetch instruction to fetch data between main memory and secondary cache without blocking the microprocessor. Between the levels of the disk and main memory, the Solaris operating system provides the `madvise() \cite{solaris:1993} system call to advise the virtu` to begin reading the specified pages currently resident on the disk into main memory. Typically, these instructions are designed to have only a small microprocessor overhead.

3.1.2 Support from the Client Server Model

In client server environments [AK97, GK94a, GK94b, Kna97c, Kna97a, PZ91, CKV93] where the microprocessor of the client is entirely independent of the server's microprocessor, non-blocking prefetches may be implemented by passing requests for data as messages to the server. In an OODBMS for example, object processing is performed by the client which relies upon the server to fetch objects or pages of objects. This leaves the client free to continue useful processing of the application.

3.1.3 Support from Batch Requests

In environments which do not offer direct support for non-blocking prefetches, prefetching may still take place. In such a system, references will occasionally result in *demand* data fetches from a level in the memory hierarchy when a requested data item is not cache resident. Demand fetches are those

which block the application's progress until the data becomes cache resident. Since it is often the overheads [HP96] of the latency (as described in chapter 2) which dominate the time for transfers between data storage devices, requests for predicted data may be batched [CFKL96] and serviced along with the demand request. Issuing multiple requests in this way effectively hides the latency of the prefetches because the additional transfer time used to prefetch data may be negligible.

This technique of deferring prefetches is analogous to the idea of *write back* [PH94] and would require similar support. Write back is a technique which defers costly write operations to high-latency storage devices, and instead buffers a number of writes in main memory, before committing them in a single operation. In this setting, a managing layer would be needed to catch all prefetch operations and buffer them until a demand read was issued. At this point, all the buffered reads could be performed, thus incurring only the latency of the demand read. Clearly, with the explicit fetch instructions discussed in section 3.1.1, special care must be taken to avoid corrupting the semantics of the application through deferral of read operations.

3.2 Issues in Prefetching Data

In most studies, prefetching has been done with little regard to the effect on the cache [CFKL95]. This lack of synergy has led to prefetching schemes which may cause performance degradation in comparison to non-prefetching systems.

This section discusses the effect of prefetching on caching and introduces a set of heuristics for their integration which has been found to be optimal [CFKL95]. In addition to integrated prefetching, there are other parameters which may affect the performance of a fetching mechanism including clustering and inter-reference time. These are also discussed.

3.2.1 Prefetching, Caching and Data Locality

The goal of prefetching, to hide latency by overlapping computation and I/O, prompts an intuitive approach to its accomplishment: prefetch the data for all references the application will make far enough in advance to hide the latency of the data storage device. However, this approach is not sufficient to reduce execution time; on the contrary, it may result in an increase. This phenomenon will now be explained in detail.

As discussed in section 3.1, caching uses the heuristic that when a data item is used, it will be re-used in the near future. This is more formally referred to as the principle of *Data locality* [PH94]. This states that applications access a relatively small portion of their address space at any instant of time. There are two types of data locality:

- *Temporal locality*: If an item is referenced, it will tend to be referenced again soon.
- *Spatial locality*: If an item is referenced, items whose addresses are close by will tend to be referenced soon.

It is clear that the intuitive approach of prefetching data for all references in an application results in many redundant prefetches: there is a strong likelihood that the prefetched data was already cache resident as a result of either temporal or spatial data locality. For each redundant prefetch, not only is there no improvement over a non-prefetching system, but the cost of issuing the wasted prefetch is incurred.

Ultimately, due to the finite storage capacities of hardware, all caches are limited in size. When a cache becomes full, further movements of data to the cache require *eviction* of data items presently in the cache back to a slower layer in the memory hierarchy. The *cache manager* uses a pre-programmed strategy to select which cache items will be evicted (the *victims*) to make way for the new data item. The lack of interaction between the cache manager, which selects victims, and

the prefetcher, which moves data to the cache, can cause performance degradation since each make their choices in isolation.

Consider the case of a cache of size n units containing n data items each of one unit in size. In this example, all n cached data items form a *working set* of frequently referenced data used by a particular part of an executing application. In addition to the microprocessor overhead of prefetching a data item, there is a space overhead. This is because space must be allocated in the cache for the new data item at the point when the prefetch is initiated. This *early prefetch* has the effect of reducing the usable size of the cache by one unit for each prefetch. Now assume that k prefetches are initiated. Since the working set of data items requires a cache of size n to ensure that no cache misses occur for the current part of the application, and only $n - k$ units of storage are available, evictions from the cache are inevitable. Until such time as the prefetched data is referenced, it is wasting valuable cache space thus forcing avoidable cache evictions and faults to the detriment of execution time. In this way, (even using accurate knowledge of future data items referenced by an application) it may not be beneficial to prefetch as far into the reference stream as possible [CKV93] since this will perturb the cached data items currently useful to the application.

In an effort to integrate prefetching and caching Cao et al [CFKL95] proposed the following rules which, when used to constrain a prefetching algorithm, would result in optimal cache management.

1. **Optimal Prefetching** – Every prefetch should bring into the cache the next item in the reference stream that is not in the cache.

2. **Optimal Replacement** – Every prefetch should discard the item whose next reference is furthest in the future.

3. **Do No Harm** – Never discard item A to prefetch item B when A will be referenced before B.

4. **First Opportunity** – Never perform a prefetch-and-replace operation when the same opera-

tions (fetching the same item and replacing the same item) could have been performed previously. The algorithm must perform each operation at the first available opportunity. This condition prevents multiple prefetches for the same items.

In addition to these, the following additional rule is necessary to ensure the effectiveness of a fetching mechanism. Every prefetch request should be serviced far enough ahead of its reference in the application to compensate for the overhead of issuing the prefetch request. Ideally, the requested data will be completely resident by the time the application references it.

These rules provide guidelines to designers of fetching mechanisms; however, conformance to these rules is complicated by a number of factors which affect the ability of a fetching mechanism to tolerate latency.

3.2.2 The Effect of Clustering on Prefetching

Clustering attempts to reduce the number of transfers across large latency barriers (eg disks and networks). This is accomplished by co-locating data items referenced by the application on larger physical units of transfer across the latency barrier. The goal behind the clustering strategy is to co-locate those data items which are referenced by the application within the same period of time.

Unfortunately, clustering is very sensitive to changes in data access patterns [MK94]. Patterns of data access can vary significantly even between invocations of the same application. Consider the case of an interactive application where the data access patterns are determined by the user's input: no single clustering is suitable for all data access patterns [GKKM92]. In such cases, clustering fails to reduce the number of transfers across large latency barriers. Prefetching has been employed as a method of hiding the latency suffered by the application when clustering fails to reduce the number of expensive I/O operations [GK94b].

However, since fetching mechanisms do not have knowledge in advance of which data items

are clustered together, unnecessary prefetches may be executed [GK94a]. These prefetches are redundant since, as a result of clustering data items onto transfer units, the requested items are likely to be cache resident.

3.2.3 Prefetch Granularity, Inter-Reference Time, and Results Ordering

The timely satisfaction of prefetch requests is dependent upon a number of factors including:

- The prefetch graularity; that is, the size of transfer unit with which prefetched data is made resident, and when (and how often) prefetches occur.
- The inter-reference time (IRT) exhibited by the application.
- The order in which prefetch requests are satisfied.

These factors require careful consideration if the fetching mechanism is to be effective. Each of these factors is now discussed in more detail.

Prefetch Granularity

While explicit fetches always prefetch data regardless of whether it is advantageous to do so, this does not imply that the transfer of data immediately follows the execution of a prefetch instruction. For example, the transfer of data may be postponed until a supporting thread to handle the prefetch request is resumed. The transfer of data may occur on the frequency of each reference, or upon each cache miss.

In an analogous situation to the frequency with which prefetch transfers occur, the unit of data transferred need not correspond to the unit directly referenced by the application. For example, although an application may directly reference objects, the unit of data prefetched may be pages of objects [CKV93, GK94a, Kna97b].

Inter-Reference Time

The elapsed time between two successive references in an application is the *inter-reference time* (IRT) between two references. This represents time spent performing compute-bound tasks which can be overlapped with I/O and so hide latency. The IRT is crucial to the fetching mechanism's ability to hide latency [GK94a]. Under conditions where the discovery of the address of the data to be made resident is dependent upon the address of a preceding data references, there may be insufficient time to make data resident before its reference in the application. This is known as a *late prefetch.*

Results Ordering

In cases where several prefetch requests are serviced at the same time, the ordering of the requests must be maintained [GK94a]. If not, then the effect on the executing application can be to consume cache space with (as yet) unneeded data and cause the application to block waiting for the prefetched data to become resident. As an example, consider the following scenario.

In a prefetching scheme which prefetches 5 references ahead, an application is predicted to make the following sequence of references: A, B, C, D, E, F. For the purpose of this example, assume that none of the data items are cache resident. Upon referencing A, the application blocks while issuing a demand fault. Along with the request for A, the fetching mechanism issues a request for B, C, D, E, F.. Under the best conditions, the data items will be delivered to the application in the order they were requested (shown in figure 3.1).

However, as a result of either the implementation of the fetching mechanism, or because each of the requested data items are in different layers of the memory hierarchy, the data may not be returned in the correct order. This results in the application blocking until its next referenced data item is resident (figure 3.2).

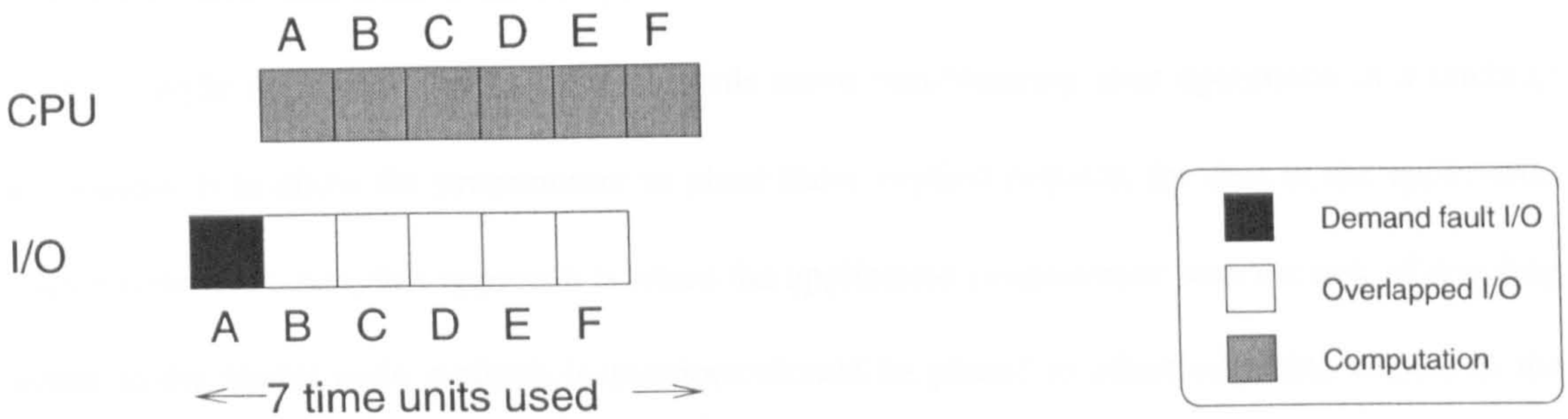

Figure 3.1: Best ordering of prefetch results

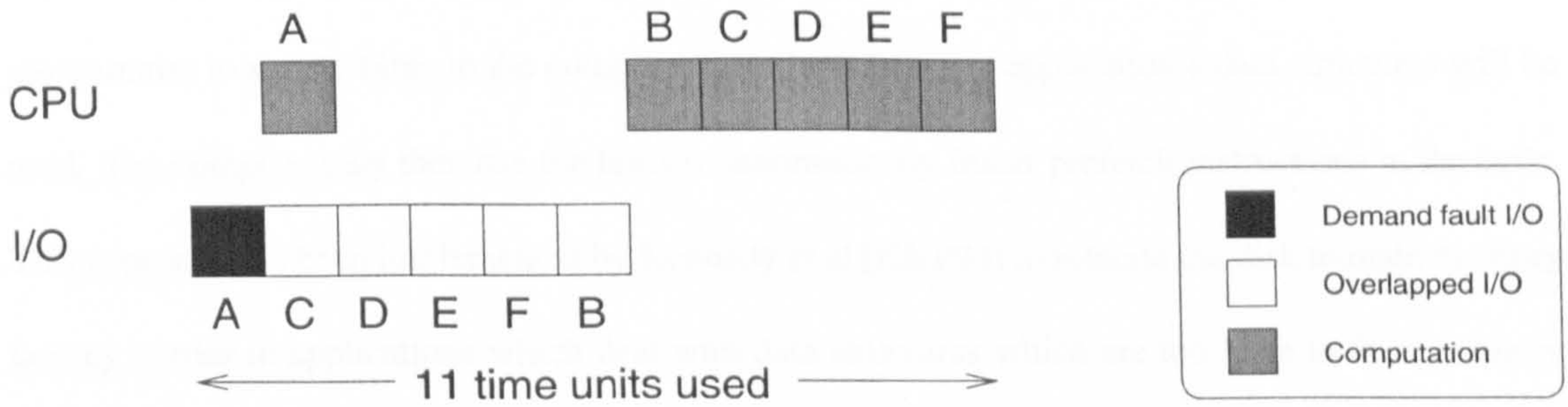

Figure 3.2: Worst ordering of prefetch results

3.3 Explicit fetching

Explicit fetching mechanisms are those which operate with the use of instructions or system calls which immediately trigger the retrieval of data upon their execution.

This section surveys the fetching mechanisms present in the literature which employ explicit fetches, highlighting their advantages and disadvantages.

The non-blocking read instructions or system calls described in section 3.1.1 and the client request messages described in section 3.1.2 have been used to implement explicit fetching mecha-

nisms. These mechanisms allow the application to perform fetching through instructions or system calls which have been inserted in the application code.

One might argue that the first step towards using non-blocking read operations in a fetching mechanism is to allow the programmer to place these explicit requests for data in the application source code. However, this approach burdens the application programmer with the task of deciding where in the source code prefetch instructions should be placed to effectively hide latency in the executing application.

3.3.1 Source-Level Compiler Hints

Instead of placing this responsibility on the application programmer, an alternative is to allow the programmer to supply hints to the compiler describing how the application's data structures will be used. The compiler may then use the hints to automatically insert prefetch instructions in the code. This approach has been implemented by Kennedy et al [KKP94] to tolerate the disk to main memory latency barrier in applications which deal with data structures which are too large to fit in memory in their entirety.

While this approach is less involved than hand-instrumenting the source code with explicit fetch instructions, it still relies upon input from the programmer in the form of data structure annotations.

3.3.2 Software Pipelining

Software pipelining [Lam88] is a fetching mechanism which has been applied to both the main memory to secondary cache latency barrier [MLG92, Mow94] as well as the disk to main memory latency barrier [MDK96]. In tolerating either latency barrier, this mechanism works best when addresses for predicted data can be computed far enough in advance to give the fetching mechanism time to make the data resident ahead of its reference, thus hiding the latency of the data transfer.

Inter-reference times of applications can affect the operation of this mechanism by constraining the amount of time available to perform prefetching. The addresses of predicted data may only be available to the fetching mechanism shortly before the data's reference in the application, and so the period of compute-bound time between references (IRT) should be great enough to tolerate the latency of the data transfers.

When applied to the main memory to secondary cache latency barrier, small IRTs do not limit the mechanism's ability to hide the effects of transfer latency. This is because the degree of latency over the main memory to secondary cache latency barrier is small, and so the address of the predicted data need only be found a short time before its reference in the application. The degree of latency over the disk to main memory latency barrier is far greater than that of the main memory to secondary cache. When applied to the disk to main memory latency barrier, small IRTs leave little time to tolerate the high latency transfers from disk.

Software pipelining is employed in prefetching schemes which operate over scientific applications characterised by iterations over large, dense matrices.

Software pipelining enables latency to be hidden by overlapping the prefetches for data needed in a future iteration with the computation of the current iteration. An example of how application source code (program fragment 3.1) is transformed to perform software pipelining is shown in program fragment 3.2.

Program 3.1 Iterative code before software pipelining.

```
for( i:= 0; i < len; i++){
  printf( ''%d'', a[i] );
}
```

A process of *loop splitting* [Mow94] is used to break the original loop (program 3.1) into the three loops shown in program 3.2. While these two programs are functionally equivalent, program 3.2 will execute in a fraction of the time of program 3.1. In the transformed program, the first

Program 3.2 Iterative code after software pipelining.

```
for( h:= 0; h < k; h++ ){
  prefetch( a[h] );
}

for( i:= 0; i < len - k; i ++ ){
  prefetch( a[i+k] );
  printf( ``%d'', a[i] );
}

for( j:= len - k; j < len; j++ ){
  printf( ``%d'', a[i] );
}
```

loop (the prolog) issues non-blocking, non-binding prefetches for the first k elements of array a. As a result, when the second loop (the steady state) begins, the computation can proceed to use the first k elements without stalling. This lookahead of k elements (the *prefetch distance*)is maintained by the steady state loop by issuing a prefetch statement for the i+kth element within iteration i. The epilogue, the third loop appears as the original loop in program 3.1. The last k iterations can be completed without stalling since the steady state loop has already prefetched the necessary data.

While this transformation is straightforward, the key parameter is how far in advance, in terms of the number of iterations, the prefetches should be scheduled. This parameter is represented in program fragment 3.2 by the constant k.

Re-use Analysis and the Impact of Data Locality

Software pipelining, as described so far, plants prefetch instructions in the code in an aggressive manner; each loop iteration will execute prefetches for every reference in the loop. In practice, many of these prefetch instructions are unnecessary as a result of data locality (see section 3.2.1)

Executing unnecessary prefetches is wasteful and their cumulative effect limits the scope for improved execution times. Mowry [Mow94] proposed re-use analysis as a solution to this problem.

By analysing the application code, references are identified which, through either temporal or spatial locality, are likely to result in cache hits. In these instances, the related prefetch for that reference may be deleted from the code.

Limitation of Software Pipelining

Software pipelining has been successfully applied to array-based source code to tolerate a range of latency barriers. However, it does not scale well to more general applications. One problem with software pipelining as a fetching mechanism is that the explicit fetches are performed without regard to the environment in which the application is executing. Even if, through consideration of a system's degree of latency, an appropriate value of the constant k can be found, the use of explicit fetches may cause performance degradation. This is a result of limitations inherent in software pipelining with explicit fetches.

With software pipelining, there is an implicit assumption that the computation required for each loop iteration will take a constant amount of time. However, constructs inside the loops such as conditional branches can cause this time to vary. Similarly, it is assumed that the time taken for data to become resident in each iteration will be constant for all iterations. This assumption is invalid since the abstraction of a flat address space is implemented using a number of physical storage layers in the memory hierarchy, each of which has a different latency barrier. Since there is no way to tell statically whether a requested data item will be resident in one particular level of the memory hierarchy, it is difficult to judge how long a data item will take to become cache resident. These problems of scheduling explicit fetch instructions in the code highlight a major drawback of this fetching mechanism: it may not be sufficient to have k set to a constant value if latency is to be hidden.

Additionally, the use of explicit fetch instructions in this mechanism makes the following incor-

rect assumptions which may cause performance degradation.

- It is assumed that the act of prefetching a data item will not disturb a cached set of data items currently being used by the application. This is not necessarily the case since the cache is of limited size. In addition to the time overhead of a prefetch instruction, there is a space overhead [CFKL95] since room must be allocated in the cache to act as the destination for the prefetched data. This limits the amount of space available in the cache to store those data items currently useful to the application. This in turn may create the need for further expensive faults and evictions from the cache.

- Software pipelining with explicit fetch instructions works under the assumption that there is no contention for system resources (eg disk bandwidth) from other external processes. This is very rarely the case. Under a loaded system, prefetch instructions can miss their target references, exposing the application to the latency penalty in addition to the wasted cost of initiating the prefetch.

3.4 Indirect fetching

Mechanisms which employ indirect fetching adopt a less rigid approach to making data resident compared to explicit fetching mechanisms. Rather than having applications issue prefetch instructions which trigger retrieval of data, applications provide hints which are received by a managing layer. The manager issues prefetches suggested by the application hints subject to criteria designed to achieve the greatest reduction in execution time. This section surveys those fetching mechanisms present in the literature which use indirect fetching.

3.4.1 A Simple Approach

The first step away from explicit fetching mechanisms is to use a prefetch hint which checks for cache residency of hinted data items before issuing prefetches for them. This approach has been used in [GK94b, Kna97b, Kna97a] to reduce the number of unnecessary prefetches, a problem common with explicit fetching techniques (section 3.3.2).

With explicit fetching, there is an assumption that the system resources are loaded such that all prefetches will be serviced in time to meet their references in the application. In practice, this is unrealistic since it is likely that other processes will be competing for memory, disk, and network bandwidth. Mowry et al [MLG92, Mow94] addressed this problem by extending their explicit fetching strategy to allow prefetches to be ignored by the operating system in cases where the memory subsystem was heavily loaded. Mowry found that, in the majority of cases, dropping prefetches in such circumstances resulted in performance improvements of their benchmark applications [MLG92]

As verified by Gerlhof and Kemper [GK94a] the benefits from prefetching strongly depend upon the timely satisfaction of prefetch requests. In particular, demand requests must not *overtake* prefetch requests. This occurs when the prefetch requests are delayed and demand requests proceed to block while making the data which was to be prefetched resident. In this case, latency will not be hidden and the overhead of initiating the prefetches will be wasted, resulting in performance degradation. A more intelligent mechanism would not permit demand fetches to overtake prefetches.

Indirect fetching permits the possibility of assigning different priorities to demand requests and prefetch requests in order to control which is performed more often. This approach has been used to bias the processing of requests in favour of demand requests [PZ91]. Although this approach appears depreciatory by allowing prefetches to miss their target references, it also has the benefit of controlling the amount of cache space available to the currently executing application instead of having large areas of the cache reserved for prefetch data.

3.4.2 Informed Approaches

Like the simple approaches presented in the previous section, integrated approaches to prefetching make use of hints provided by executing applications which disclose their future data access requirements to a managing layer. Unlike the simple approaches however, decisions on which hints to issue prefetches for, and the time at which they are issued, are made in the presence of knowledge of the current cache utilisation and the competition for system resources. This knowledge is used to guide both prefetching and cache management over multiple competing processes.

Integrated application controlled prefetching, caching and disk scheduling [CFKL96] attempts to manage system resources in the presence of several competing prefetching applications. Two-level cache management is used to share cache space among all applications and let each application control the prefetching and caching decisions over its own cache area. The controlled aggressive prefetching algorithm presented in full in [CFKL95] conforms to the rules for integrated prefetching and caching (presented in section 3.2.1) is used to give near optimal performance of the cache in the presence of prefetch hints.

While the approach taken by Cao et al [CFKL96] has been shown to be near optimal in its cache management, it does not compensate for the effects of constrained system resources on prefetching. Transparent informed prefetching with temporal overload estimators (TIPTOE) [TPG97] uses a cost-benefit analysis to judge the impact that decisions on caching and prefetching data items will have on execution time. This cost benefit analysis takes into account the load on data storage devices and the effect it will have in delaying prefetch response time.

Data storage devices do not exhibit infinite parallelism. That is, they can only cope with a finite number of requests in parallel. If a prefetching mechanism should issue prefetches to a data storage device such as a single disk, the requests will be bottle-necked in the device manager controlling the disk. This will result in starvation of the prefetch requests. The solution to this as proposed

in [TPG97] is to prefetch deep into the reference stream in order to tolerate the latency of both the data storage device and the queue of prefetch requests. Employing deep prefetching with cost benefit analysis which takes into account the load on data storage devices has been shown to result in better performance [TPG97] than integrated prefetching and caching [CFKL95].

3.5 Conclusions on Making Predicted Data Resident

This chapter discussed the mechanisms used to make predicted data resident. These mechanisms were broadly classified into those which use explicit fetching and those which use indirect fetching. The primary difference between the two being that although, as in explicit prefetching, data is made resident in parallel with the application's execution, indirect fetching mechanisms include checks to ensure that prefetching is likely to improve performance.

There are many factors which can lead to explicit fetching mechanisms degrading performance of applications, since the prefetch will be performed regardless of:

- the presence of requested items in the cache
- the demand for memory, disk, or network bandwidth
- the effect of multi-user or multi-threaded loads on the service times for prefetches.

While it is clear that the overheads of indirect fetching mechanisms are higher than those of explicit mechanisms, it is highly likely that the additional overhead costs are more than covered by the resulting improvements in execution time, if not by reduced cache misses.

Chapter 4

Predicting Data Requirements

As discussed in section 2.3.2, prefetching schemes require prediction mechanisms to predict data items which are likely to be required by an executing application program prior to the use of those data items. Once this information is available, it can be used by the mechanisms described in the previous chapter to make the predicted data resident ahead of its use by the application program. In this way, the latency of data retrieval is tolerated and execution times may be reduced.

The absence of either a sufficiently broad survey of prediction mechanisms, or a system for their classification obscures recognition of the fundamental concepts involved in prediction. By introducing such a classification, prediction mechanisms from radically different application areas can be understood within a single framework.

This chapter discusses the fundamental concepts involved in the prediction of an application program's data requirements. A system of classification is then proposed upon which a taxonomy of extant prediction mechanisms is built which spans multiple application areas.

4.1 Requirements for Prediction

As components of a prefetching scheme, prediction mechanisms share many of the requirements for prefetching introduced in section 2.3.2. In particular, any prediction mechanism must satisfy the implementation requirement of minimal *prediction overhead.*

In addition to this, fundamental requirements of the prediction mechanism's functionality must be satisfied in order to reduce the impact on execution performance. These include: high *prediction accuracy*, long *prediction lookaheads* and large *prediction coverages*. These requirements are now discussed in further detail.

4.1.1 Minimal prediction overhead

In those prediction mechanisms which perform prediction in parallel with an executing application program, the cost of performing prediction is termed the prediction overhead. This overhead encompasses the additional expenses in terms of memory footprint, I/O traffic, or CPU processing incurred as a result of the prediction mechanism's operation.

If prefetching is to be effective in reducing execution time, any additional cost of predicting future data requirements should be smaller than the benefit brought by prefetching the data, and ideally should be as small as possible. The reasoning behind this requirement is explained by the following example. If the time taken to predict n future data accesses is greater than the latency of suffering n cache misses, then the prefetching system will increase the execution time of the application program compared to a non-prefetching system. Even if the cost of predicting the next n accesses were less than the cost of n cache misses, it is unreasonable to assume that n references will result in n cache misses as a result of data locality (section 3.2.1). In the presence of data locality, where predicted data may already be cache resident, even a predictor cost which is smaller than the latency penalty can result in a performance degradation [CFKL95].

4.1.2 Accurate prediction

The sequence of data items predicted should accurately mirror the sequence of data items used by the application program. Each incorrectly predicted data item increases execution time compared to a non-prefetching system. This increase can be attributed to:

- The CPU-bound cost of processing the initiated requests for unneeded data.
- The cost of the additional faults and evictions to the cache as a result of *cache pollution* [Smi78] where cache space is consumed by unneeded data items. This necessitates additional cache evictions (see section 2.2.1).

4.1.3 Prediction Lookahead

In order to be effective, prefetching schemes must be able to predict far enough into the future reference stream of an application program to be able to hide the latency of data retrieval. The earlier a prediction can be made, the earlier a prefetch can be initiated and so the greater the scope for hiding latency. If prediction of a particular data item's use can only be performed shortly before its use in the application program, it is unlikely that the latency of the data access will be fully hidden.

4.1.4 Prediction Coverage

The prediction coverage describes the percentage of the application's data requirements which could be predicted, regardless of whether those predictions were accurate. This measure describes the coverage of the prediction mechanism.

4.2 Prediction Environments

There are a number of prediction mechanisms found in the literature, each of which predicts the behaviour of an application in terms of its data requirements. The mechanisms obtain the information necessary to perform prediction from a *prediction environment* (discussed fully in section 4.3). The richness of information available in the prediction environment greatly influences the degree to which knowledge of the application can be used in a prediction environment.

Each of the prediction mechanisms found in the literature is based upon the hypothesis that the characteristics of an application program's code, data, or run-time behaviour can be used to predict its behaviour in terms of its data requirements. These characteristics may be recognised manually by the designer of the prediction mechanism, or automatically by the prediction mechanism itself. For example, the designer of a prefetching scheme for file systems may notice that the bulk of all disc block accesses follow a sequential pattern [1] and decide to exploit this observed characteristic of the file system in the prefetching scheme's prediction mechanism. This might involve implementing a prediction mechanism that uses a fixed strategy of predicting block $n + 1$ when block n is accessed [KE90]. Alternatively, the prediction mechanism itself can automatically recognise characteristics of applications which can be used to predict future data accesses. For example, a prediction mechanism for a virtual memory pre-pager [Tri76] may use a sequence of page faults as the key into a pattern memory of page access sequences [CKV93] to predict those which will follow.

4.3 Prediction Mechanisms and Portability

Studies have already shown prediction mechanisms being applied to different application areas [GK94a, MDK96, LM96]. Each of these application areas present prediction environments which support the

[1] Typically, files are accessed sequentially in their entirety.

needs of a particular prediction mechanism.

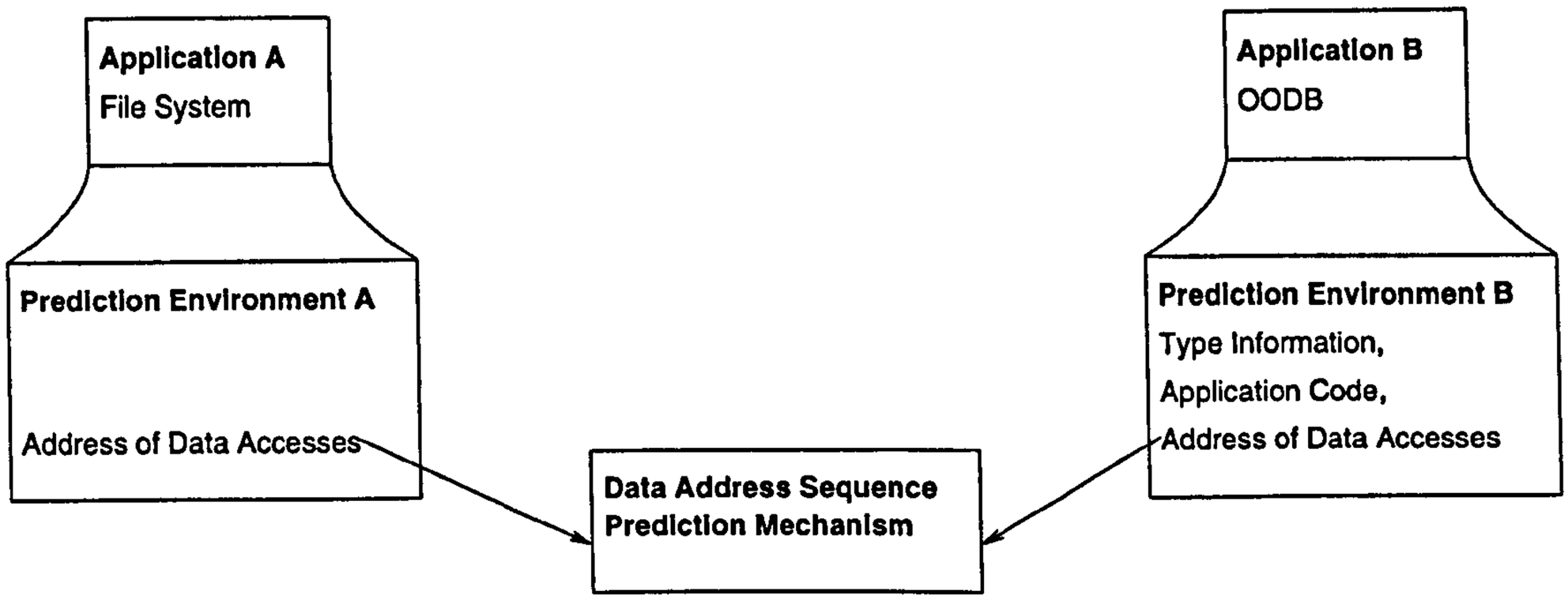

Figure 4.1: It is the prediction environment which supports prediction mechanisms. This view has enabled prediction mechanisms to be ported to different applications

Prediction mechanisms which do not rely upon information specific to one prediction environment are the most portable. Those prediction mechanisms which rely upon highly application-specific information from the prediction environment are less portable to different application areas.

Figure 4.1 shows how prediction mechanisms can be ported to other applications provided they offer a prediction environment suitable to the prediction mechanism. The figure depicts the example of a prediction mechanism which relies upon pattern matching over addresses accessed during the execution of an application program. The prediction mechanism requires a simple prediction environment like that provided by application A, a file system which services requests for files. This environment provides only the addresses of data items requested. This prediction mechanism could also be applied to an application with a richer prediction environment such as that of application B, an OODB application.

In contrast, consider a prediction mechanism which utilises information of the code or schema of the data used in the application. This requires a prediction environment which provides this informa-

tion. Consequently, the prediction mechanism cannot be applied to simpler prediction environment.

4.4 A System of Classification

An important step in being able to provide a unifying set of concepts for prediction is the creation of a classification system upon which a taxonomy of prediction mechanisms can then be built. This section proposes dimensions along which prediction mechanisms can be usefully classified for the purposes of comparison.

Despite their origins in different application areas and addressing different latency barriers, prediction mechanisms divide naturally along two orthogonal dimensions: *prediction perspective* and *time of prediction.*

4.4.1 Prediction Perspective

Fundamentally, all prediction mechanisms are based upon assumptions made about the characteristics of an application. These assumptions can be derived either from recognition of explicitly codified dependencies inherent in the application, or through tacit knowledge gained through observation of the application's behaviour patterns.

The difference in these approaches can be expressed as the difference between knowing why an application exhibits a particular behaviour, and simply observing that it does exhibit (or tend to exhibit) a particular pattern of behaviour.

The prediction perspective is constrained by the prediction environment (section 4.3). It is impossible for a prediction mechanism using codified knowledge of an application to operate in an environment which doesn't supply access to that codified information.

Codified Knowledge Perspective

In the case of assumptions which exploit codified dependencies, the behaviour of the application may be predictable as a result of dependencies or constraints asserted by the application program.

The use of codified dependency assumptions is made possible by prediction environments which afford access to the types [CK89], data sets [Kna99], or code [MLG92, MDK96, KKP94, LM96] of an application program.

Tacit Knowledge Perspective

In the case of assumptions based on tacit knowledge of the application, the behaviour of the application may be predicted on the basis of its observed operation. The quality of the prediction in this case depends upon the degree to which the observations accurately represent all aspects of application behaviour [PZ91, CKV93].

4.4.2 Time of Prediction

Although one may expect a fetching mechanism to operate as the application executes, this is not so with prediction mechanisms. The *time of prediction* is the point at which the prediction environment is analysed and predictions of data requirements are made. This process may be performed when the application is not running (*static prediction*) or as the application runs (*dynamic prediction*).

Static Prediction

By performing all prediction off-line, static prediction mechanisms incur a minimal prediction overhead (section 4.1.1). Since prediction is performed off-line, this allows more expensive analyses of the information in the prediction environment.

Within the classification of static predictors, there exist code-based and data-based prediction

mechanisms.

Dynamic Prediction

Dynamic prediction mechanisms analyse the prediction environment to provide predictions as the application program executes. This carries a higher overhead than static prediction mechanisms.

4.5 A Taxonomy of Prediction Mechanisms

The system of classification proposed in section 4.4 enables the construction of a taxonomy of prediction mechanisms present in the literature. This taxonomy is presented here.

	Static	Dynamic
Codified knowledge	[CK89],[KKP94],[Kna97b], [LD92],[LM96],[Mow94], [MDK96],[Tri76] [KGM91]	–
Tacit knowledge	[Bes95],[GK94b],[GA94], [GAN93],[KE90]	[AK97],[BKW94],[CFKL96], [CKV93],[MK94],[PZ91]

This section presents the many sub-genera of prediction mechanisms which lie along the dimensions of prediction perspective and time of prediction.

4.5.1 Static Code-based Prediction Mechanisms

Loop splitting [MLG92, Mow94] is used in conjunction with the software pipelining mechanisms (introduced in section 3.3.2) to perform prefetching in array-based scientific applications. Loops in the source code which iterate over arrays are analysed and re-written to allow prefetches to be scheduled in the code. Each loop is split into three separate loops to:

1. Prefetch the data for the first k iterations of the original loop. This is used to tolerate the latency of the fetching data in the first k loop iterations.

2. Process the data of iteration i and issue prefetches for the data used in iteration $i+k$. The data to be processed in the first k iterations is prefetched by the first loop. This loop terminates k iterations before the end of the original loop.

3. Access the data needed by the final k iterations of the original loop. These will have been prefetched by the previous loop.

Loop splitting predicts an application's future references, however, due to the principle of data locality, not all references will result in a cache miss. Since issuing prefetches for already resident data incurs a run-time cost (section 3.2.1) it is desirable for prefetches to be scheduled only for those references which are likely to cause cache misses. To this end, Mowry et al use re-use analysis to identify these references for which no prefetch should be scheduled. This works well for small loops, however with larger loops, re-use analysis does not cope well since whether a reference will be a cache miss depends upon the size of the cache. Wilson et al [WKM94] made the observation that it is impossible to tell statically whether a reference will cause a cache miss. As an attempt to address this *memory size relativity*, Horowitz [HMMS95, HMMS98] introduced a informing load operations to Mowry's original compiler-based prefetching algorithm [MLG92] to allow the prefetching mechanism to adapt to the cache state. Specifically, the code produced by the compiler prefetches using the informing memory operations to record the number of times prefetch requests were unnecessary, since the data was already cache resident. If the number of cache hits upon prefetches increased above a fixed limit, the application code jumps to a non-prefetching version of the loop. In this way dynamic information could be used to produce a largely static prediction mechanism which was still able to react to dynamic events.

4.5.2 Static Data-based Prediction Mechanisms

In contrast to static code-based prediction mechanisms, static data-based mechanisms enable the prediction of only those references which will result in cache misses. As discussed in chapter 3, this greatly simplifies the task of prefetching.

Knafla [Kna97b, Kna97c, Kna97a] designed and implemented such a prediction mechanism in a prefetching scheme for OODBMSs where the latency penalty incurred by accessing non-resident pages is considerable. In this environment, objects are grouped on pages which are transferred between the client and server. When the database is off-line, every object on each page of the database is scanned for references to objects located on other pages. Such objects are termed *outward referring objects* (OROs). As the application processes an ORO, a reference may be made to the object on an external page. If the page containing the referenced object is not resident, an expensive page fault will occur.

In order to tolerate the cost of the page fault, an object further up the chain of references from the ORO is marked as a *prefetch start object* (PSO). When the application processes this object, a prefetch is initiated for the page containing the object referenced in the ORO.

The process of choosing an object to be tagged as the PSO is complicated by the following problems.

- The *prefetch object distance* (POD) is the number of objects lying in the path between the PSO and the ORO. The size of the POD should be large enough so that the time taken for the application to process all objects between the PSO and ORO is as great as the time taken for the page server to fault the page containing the object referenced by the ORO. In this way, the time for a page fault must be related to the time for object processing. However, the degree of object processing varies between applications, so the POD should be tuned to the application and the latency of the database's page server.

- PSOs trigger prefetches for pages under the assumption that a particular chain of objects will be traversed from the PSO. In this way, each PSO should refer to a single ORO. However, it is possible for two separate OROs to have the same PSO. In such cases, there is an ambiguity over which path to assume will be taken and hence which page to prefetch. The problem is illustrated in figure 4.2. Through analysis of page A, three OROs are identified: A3, A5, and A7, which refer to B1, C4, and D1 respectively. Assuming that the POD is required to be 2 objects in length to hide the latency of a page fetch, the PSO for each of A3, A4, and A7 is A1. A1 can therefore no longer be used to identify a page which will definitely be used in the near future.

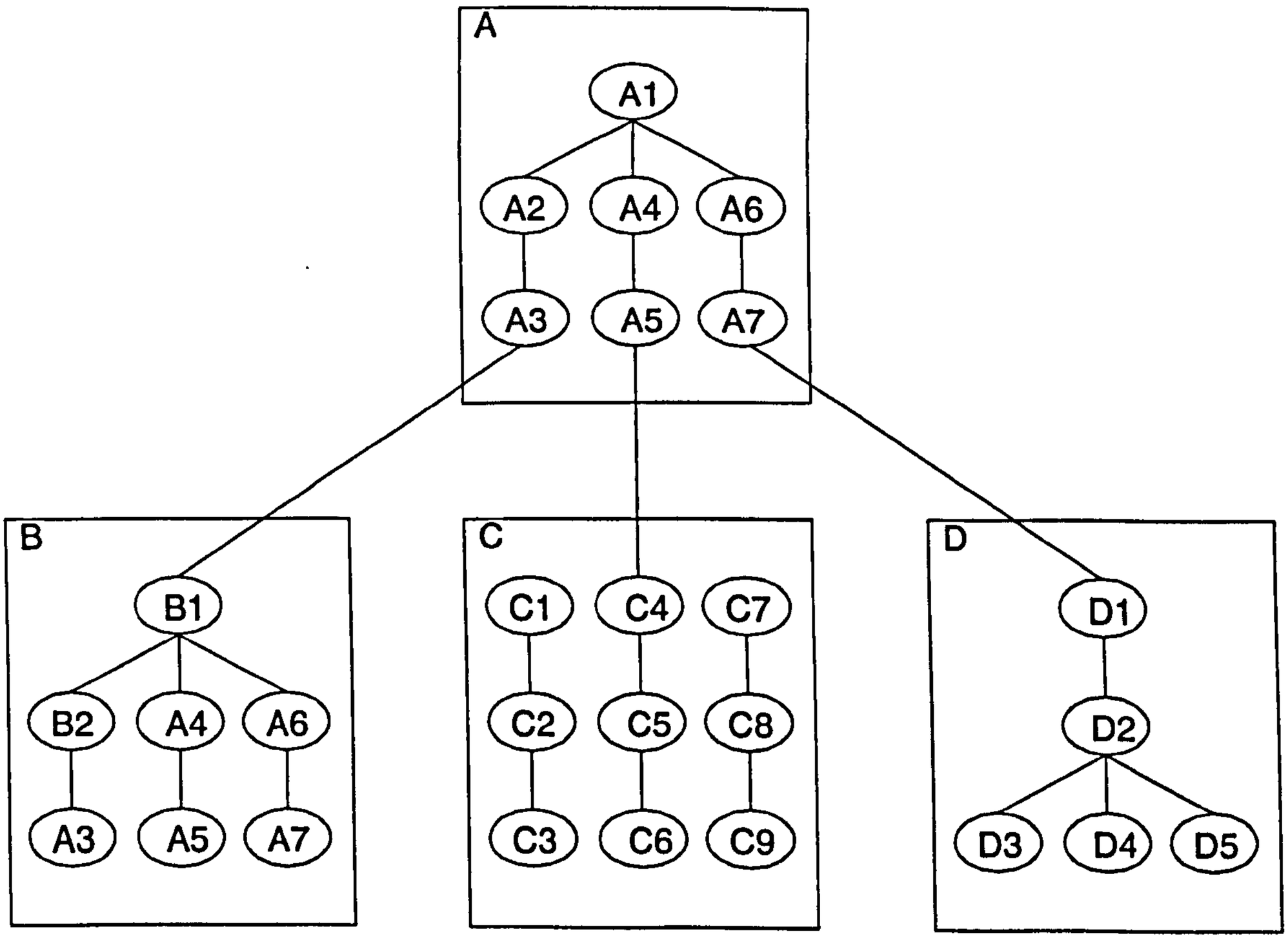

Figure 4.2: Choosing a prefetch start object

In order to cope with the three possible paths which may be taken from the PSO to an ORO,

prefetches can be issued for pages B, C, and D. However, this would result in cache pollution (section 4.1.2). An alternative approach is to shorten the POD such that there is an non-branching path of objects between the PSO and ORO. In the example shown in figure 4.2, A2 would be the PSO for A3, A4 for A5, and A6 for A7. This solution sacrifices some of the latency time which would have been hidden by a longer POD in favour of more accurate prediction to reduce cache pollution.

In databases which contain many branch objects, it is likely that the the length of unbranched object chains will be less than the latency of page access. Considering the worst case, when the ORO itself contains a number of references to objects located on more than one external page. Under such a scenario, it is be prudent to ignore prefetches, since no latency can be hidden, and it cannot be determined statically which branch will be taken from the ORO.

Knafla's solution [Kna97b] to this problem is to perform a period of monitored execution [Kna98] in which the frequency with which each object reference is traversed is used in choosing a path of objects from a branching PSO. For each object in the database, the frequency with which each of the object references is traversed is stored as a probabilistic weight. When several OROs nominate the same object as their PSO, the weights of the references leading to the OROs are compared. The object reference with the highest weight is used to select which external page will be prefetched.

There are disadvantages to this mechanism. Firstly, it doesn't adapt well handle to mutations of the database. Since the prediction is performed off-line, on the basis of the current database snapshot, the changes made to the database by the running program can invalidate the predictions. This is particularly a problem when it comes to variable references in method code. Secondly, although it can limit the number of possible number of references which might result in an expensive fault, it still can't say which of those references will result in an expensive fault, since the page in question may already be resident.

4.5.3 Dynamic Predictors

In contrast to the prediction mechanisms of the previous section, the mechanisms described in this section perform prediction as the application executes. Typically, run-time prediction mechanisms consist of a software component which, while separate from the application, executes in parallel with the application to predict its future data accesses.

Run-time mechanisms in the literature can be classified as belonging to one of the following classes of predictor:

- *strategy-based* predictors [MJLF84, AK97, KE90] use a pre-programmed, fixed strategy based on the behaviour of the application as perceived by the designer of the prefetching scheme.

- *structure-based* predictors [CK89, Cha89, KGM91] use information of the application's data types and the nature of operations over the data them to predict future data accesses.

- *training-based* predictors [GK94b, GAN93, CKV93, PZ91] use data access patterns obtained either earlier in the current invocation of the application or from some previous invocation(s) to predict the application's future data accesses.

  The predictor may operate in a clamped *training mode* in which the application's data accesses are analysed and patterns are formed and stored in a memory which can be accessed efficiently when the predictor reverts to *prediction mode*.

  Alternatively, the process of training and prediction may be on-going where the application's execution is constantly being monitored and adjustments made to the pattern memory.

Since run-time predictors execute in parallel with their target application, the mechanism's prediction overhead (section 4.1.1) is crucial in delivering a prefetching scheme which is effective in reducing execution time. As a result, the design of data structures and algorithms used in predic-

tion plays a major part in the prefetching scheme's overall effectiveness. Gerlhof et al [GKKM92] categorise run-time prediction mechanisms as either:

- *fast predictors* which employ a simple table lookup or follow a simple pre-programmed strategy. These are often low-cost training-based mechanisms which operate mainly in a prediction mode and occasionally revert to a training mode to update a table of data access patterns. Most strategy-based predictors also fall into the category of fast predictors.

- *slow predictors* which may require a considerable degree of computation to predict future references. This describes many training-based mechanisms which simultaneously predict data accesses while dynamically updating pattern memories of data accesses as programs execute. Structure-based techniques also fall into this category.

The trade off between fast and slow predictors is one of prediction overhead versus accuracy and adaptability to change in data access patterns. While fast predictors incur only a small run-time overhead to impede the executing application, they are, in a number of application areas, not very accurate or do not adapt quickly enough to changes in data access patterns. Conversely, slow predictors offer accurate prediction of data accesses and cope with changes in data access patterns, although their run-time overhead can be prohibitively expensive [PZ91, CKV93]. A recurrent theme in this and the preceding chapters has been the cost of inaccurate prediction in terms of memory space and microprocessor time. Whether slow predictors perform more favourably than fast predictors depends upon the expense of the prediction overhead (in terms of both space and time) of a slow predictor compared to the impact on execution time of a fast predictor's inaccurate predictions.

This section discusses runtime prediction mechanisms found in the literature which can be classified as either strategy-based, structure-based, or training-based. In doing so, this chapter aims to expose the benefits and shortcomings of each predictor class and so present the reader with an

appreciation of both how the mechanisms work and to which application areas they are best suited.

Strategy-Based Predictors

Prediction in strategy-based predictors is controlled by a fixed, pre-programmed strategy. Particular strategies are chosen by the designers of prefetching schemes in response to their observations on, or intuition of the characteristics of the application. In this way, strategy-based predictors must be tailored not only to their application area (eg file systems, OODBMSs, Translation Lookaside Buffers) but to the access patterns most frequently encountered.

For example, consider sequential one-block-lookahead [Jos70, MJLF84]. This strategy-based predictor has been commonly used in prefetching schemes for file systems. Designers of prefetching schemes have analysed traces of file system requests and found that disk block requests predominantly follow a sequential ordering. This observation on the part of the designer has been exploited in sequential one-block lookahead predictors. These predictors use the simple strategy of predicting that block $n + 1$ will be the next block accessed upon visiting block i. A prefetch is then issued for the predicted block, with the new block being transferred to a fixed slot in the cache which is used only for storing prefetched blocks in order to minimise perturbation of the rest of the cache. As with most of the simpler strategy-based predictors, the chief advantage is the low prediction overhead. In terms of memory space consumption, this predictor consumes only one more slot in the cache than a non-prefetching system. Similarly, the microprocessor time required is small since a simple increment of the current block address is all that is involved.

Kotz and Ellis [KE90] extend the simple sequential one-block-lookahead predictor by observing that the prefetches often resulted in cache slots being reserved, but not filled by the time they were accessed. This caused the application to block since the space in the cache had been reserved, but the block I/O to the cache had not been completed. Through experiments, favourable results were

found [KE90] for a number of benchmark applications where the sequential one-block-lookahead strategy was modified to increase the prefetch lookahead distance. For example, when visiting block n, block $n+i$ is prefetched where i is a constant large enough to hide the latency of data I/O.

The strategy-based predictors discussed so far have been from the application area of file systems where the predominant access pattern is that of sequential block accesses. With the same application area, but with different access patterns, the simple one-block-lookahead strategy discussed so far is inadequate. In adapting a sequential lookahead predictor to pre-paging for array-based programs [Jos70] the effect of non-sequential access patterns must be considered. In a matrix multiplication program, for example, calls are made alternately to elements of two different matrices. In this pathological case, the use of one-block-lookahead results in the prefetch cache slot being repeatedly overwritten with the wrong block and thus, the prediction is always incorrect. The solution to this problem [Jos70] is to increase the number of cache slots reserved for prefetching and to prevent recently prefetched blocks from being evicted before their reference in the application.

The strategy-based predictors discussed so far have relied upon a simple pre-programmed strategy. The data items are predicted in a uniform manner using fixed-length increments to data addresses. This rigid prediction mechanism makes no allowances for changes in data access patterns, or the state of the cache in terms of its size and set of resident data items, or the contention for cache space caused by concurrently executing threads. However, the use of a pre-programmed strategy does not preclude the possibility of more flexible strategy-based predictors. The prediction strategy used in selective eager object faulting (SEOF) [AK97] is a good example of a flexible strategy-based predictor.

SEOF is an object prefetching scheme for those OODBMSs based on the dual buffer architecture. In this case, the designers of the system based their predictor's strategy on their belief that, under an appropriate clustering for a particular execution:

1. Pages of objects which are referenced repeatedly over a long period of execution contain objects which are useful to the current execution. As a result, such pages should have all their objects prefetched to the object cache.

2. Pages of objects which are referenced frequently over a short burst of execution, but relatively infrequently over a longer period of execution contain only a few objects of use to the execution. Objects on these pages should be fetched to the object cache on a per-object basis.

This strategy relies upon an appropriate clustering of objects onto pages for a given query and as a result the approach to prediction is rather more speculative. The assumption is that the prolonged series of objects accesses to the same page indicate that a page is a good candidate for prefetching of all its objects. However, it may be either that the accesses were to the same object each time, or to a very few of the objects on the page. In such cases, prefetching all the objects on that page may prove to be a waste of time, since they may not be required by the executing application.

The predictor strategy for SEOF uses two FIFO queues, S_{in} and S_{out} of length $thresh_{in}$ and $thresh_{out}$ respectively (see figure 4.3). Upon an object request which misses in the object cache, the page containing the object has its identifier inserted in S_{in} provided the page id is not present in either S_{in} or S_{out}. When S_{in} becomes larger than $thresh_{in}$, the first-come entry of S_{in} is moved to S_{out}. S_{out} keeps its length in the same way. If the page containing the missing object is found in S_{out}, it is considered a candidate for eager fetching of its remaining non-resident objects. Only the pages present in S_{out} which have objects referenced are candidates for prefetching. The size of the queues $thresh_{in}$ and $thresh_{out}$ control the sensitivity of the selection of candidate pages.

This is a low-cost prediction mechanism: the simple queue system used to find candidates for prefetching does not require significant amounts of processing. While it is low cost, it is evident that its prediction accuracy is dependent upon the object to page clustering and the size of the queues. As such, this prediction mechanism is typical of strategy-based predictors in that they are light-weight,

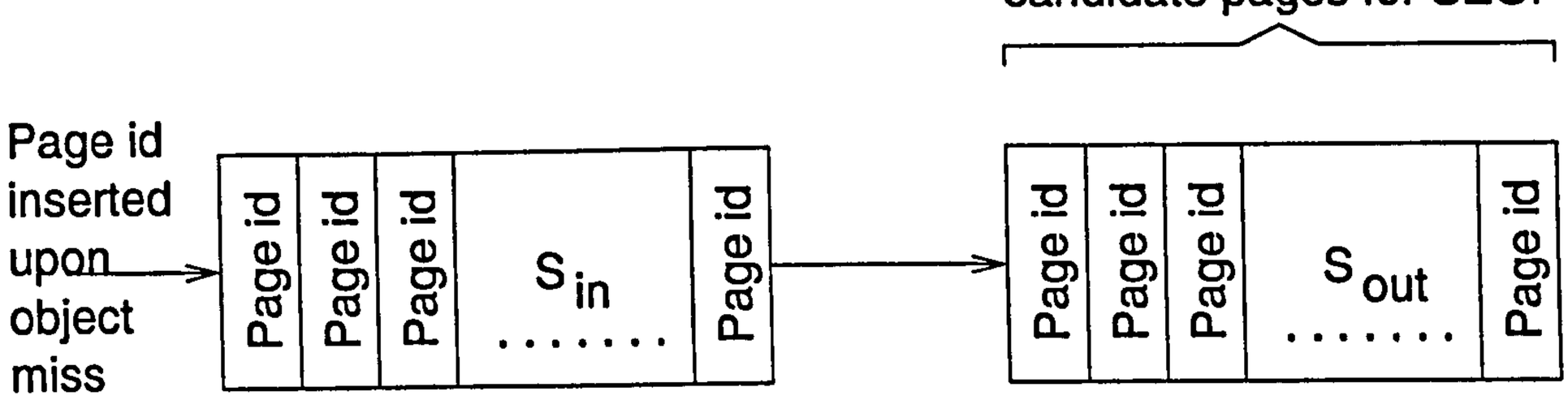

Figure 4.3: Selective eager object faulting

but do not automatically adapt to changing access patterns.

Structure-Based Predictors

Structure-based predictors are mostly found in OODBMS prefetching schemes. These predictors utilise knowledge of the database schema including its data types to form predictions.

Keller et al [KGM91] designed a structure-based predictor for use in OODBMS where references to complex objects are pre-emptively resolved. An *assembly operator* assembles complex objects in main memory in time for their traversal by a query. The *component iterator* uses structural and statistical information concerning the schema to predict the required objects and optimise query performance. The task of finding an ordering of assembly for complex object parts is specific to each query and is structure and type dependent. To cope with this, templates containing the information used by the component iterator are created by the database administrator. The template takes the form of the complex object which is to be traversed by the query. The object references in the complex object are augmented to indicate the degree of sharing between objects and predicate selectivity. The primary disadvantage of this mechanism is reliance upon the database administrator to create the templates for common queries over the database.

Chang and Katz [CK89, Cha89] introduced a prediction mechanism for OODBMSs which, like

the one introduced in [KGM91], is reliant upon human intervention to guide the prediction process. Upon starting a new database session, the user supplies information on the main path to be taken through the object graph. For example the user may identify particular fields of a class to show which fields of objects (and the order of those field references) will be followed in the application. A prefetching mechanism is then used to make these resident ahead of their access in the application.

Training-Based Predictors

Training-based predictors predict future application references or faults on the basis of past data access patterns. This requires methods for:

- adding data access patterns to memories which can be maintained as data access patterns change.
- recognising the start of previously encountered access patterns in the application's reference stream.
- producing predicted references in response to recognised patterns.

Training-based predictors provide one of two models of operation.

1. The predictor may perform prediction and training in parallel, thus supplying predictions in a *continuous* manner.
2. A training *phase* (perhaps performed off-line) allows for the consolidation of new data access patterns into the memory of access patterns used in a prediction phase.

Examples of both continuous predictors [GAN93, CKV93] and phased predictors [GA94, PZ91] are now presented.

Multiple Prefetch Adaptive Disk Caching [GAN93] offers prediction in the face of changing data access patterns without the need to perform separate training phases. The predictor's memory

is represented by a table, with entries corresponding to clusters of sequential blocks on the disk. The clusters themselves correspond to slots in the disk cache. The table is indexed by cluster number. Each entry in the table contains the number of the next cluster predicted, and an integer weight representing the certainty with which the predicted cluster is accurate. As the application executes, the weights on the predicted clusters are adjusted in retrospect. If the predicted cluster was accessed by the application, then the integer weight is incremented, if it was incorrect, it is either decremented while still greater than zero, otherwise the next predicted cluster is changed to the current cluster.

The principles of data compression can also be applied to predicting data access patterns. The principle, discussed in [CKV93], states that compressors which build a dynamic probability distribution over input data can be used to predict page accesses in an OODBMS given a sequence of page faults. A number of data compression algorithms were adapted in [CKV93] to predict page accesses. The most viable of these prediction algorithms in terms of time and space complexity were based on Markov chain predictors [MT93]. These predictors draw their predictions on the basis of a window of recent data accesses.

A prediction mechanism which relies upon the periodic analysis of previously collected traces of system execution is presented in [GA94]. Here, the prediction mechanism predicts which files will be opened in the near future given the position of the current file in an undirected graph. The graph represents the files as nodes in the graph, with the most likely successors to the current file being clustered around the current node as its nearest neighbours. The links between the nodes are biased to reflect the probability of opening a particular file after the current one. This biasing of the links is performed in a training phase in which the gathered execution traces are analysed. Assuming that k prefetches may be issued, then the top k biased descendents of the current node are chosen for prefetching.

Prediction mechanisms which use separate training and prediction phases are also used in OODBMSs. The FIDO [PZ91] system relies upon the client to collect reference traces for each session and for each *access context*. Access contexts are used by the predictor to isolate the references caused of different sources in order to make training easier. The training mode processes each context's reference trace normally (but not necessarily) off-line, between sessions and incrementally improves a lossy pattern memory. The lossy characteristic of the pattern memory allows minor variations in patterns over time to be ignored, and only significant changes to result in modification of the memory. In addition, lossy memory allows predictions to be made even if the access pattern does not exactly match the remembered pattern. Prediction is performed by parsing the access sequence of individual contexts for recognised keys to the pattern memory. Upon a successful match of a recognised key in the access sequence, the rest of the pattern is recalled from the pattern memory and prefetches issued accordingly.

4.6 Cross Cutting Issues

There are a number of issues which affect the performance of prediction mechanisms regardless of their position in the taxonomy. These are discussed here.

4.6.1 Unit of Prediction

Across the prediction mechanisms presented in the taxonomy, the nature of what actually gets predicted depends upon the different contexts of those prediction mechanisms. This far, the term "data requirements" has been used to describe the data which will be used when an application program executes on a computer system. This is a general term. In fact, the *unit of prediction* varies between the mechanisms in the literature. This is partially dependent on the type of information afforded by the prediction environment.

4.6.2 Dependencies Upon Data

As discussed in section 4.3, the portability of mechanisms is dependent upon the information available from the prediction environment. In addition to these constraints, the ability to port one prediction mechanism to another application has been seen to depend on the nature of the data [GK94a].

The pointer chasing problem [LM96] complicates prediction mechanisms which attempt to perform loop unrolling based prediction on code which uses data structures which are not contiguous in their representation. The problem is one of address discovery: the information necessary to form predictions is stored in the very data which is the target of the predictions. Work has been done in exploiting interfaces to change the underlying representation of data to move away from structures which are prone to the pointer chasing problem. However, since recursive data structures were probably chosen for performance or scalability reasons, it may be unwise to change the representation in this way.

4.6.3 Maintenance and Adaptability

Application behaviour is a product of application programming and application data. Recall that prediction mechanisms rely upon assumptions of application behaviour (section 4.4.1). When changes in the application program or the data used in the application change, this behaviour will inevitably change. The ability of the prediction mechanism to adapt to these changes is important.

4.7 Summary and Conclusions

This chapter introduced concepts used to classify prediction mechanisms present in the literature. A taxonomy of prediction mechanisms under these classifications was then given.

The main dimensions along which the classifications were developed were prediction perspective and time of prediction. Prediction perspective can either be based on tacit knowledge of an

application, or codified knowledge of an application. Time of prediction, relating to the time at which information from the prediction environment is used to form predictions can either be static and performed off-line or dynamic and performed as the application runs.

This chapter introduces lightweight run-time prediction mechanisms as well as static predictors which are free of any run-time overhead. The benefits of using more costly prediction mechanisms with more accurate prediction versus the lightweight prediction schemes depends partially upon the size of the latency barrier, since this is the time at which prediction is most often done in a run-time system. However, too heavy a prediction scheme can cause paging problems due to the size of the housekeeping structures used predict future data accesses.

Even though it is possible to perform predictions statically using the tacit knowledge perspective, (section 4.5.2), prediction is performed on the basis of the pages which are likely to result in a fault. Due to the combined effect of caching and clustering however, page faults cannot be predicted, only their possibility. With dynamic prediction however, it is possible to perform prediction from the tacit knowledge perspective in such a way that it is the page faults which are predicted.

One of the main disadvantages of static techniques is their inability to predict the actual page faults, since any analysis takes place away from a running system, and so information of current cache utilisation is not available. Knafla attempts to address this problem by trying to examine the problem from the storage layer instead of the application's references. Even so, the absence of a run-time environment in which cache size and utilisation is known makes prediction of expensive cache misses expensive. This presents a significant problem, since it has been found that in practice [KGM91], the frequent issuing of prefetches for cache resident data items causes a significant performance degradation.

It has been shown that codified knowledge perspective code-based prediction mechanisms provide 100% prediction accuracy, but have very limited prediction lookaheads due to problems such

as pointer-chasing. As a result, they are effective in situations where the degree of latency to be tolerated is small. In contrast, mechanisms based on a tacit knowledge perspective such as the training-based mechanisms presented section 4.5.3 can provide extremely long prediction lookaheads since they are independent of the application code. However, with these mechanisms, prediction accuracy is highly dependent upon application locality, and the clustering of data. Generally, the more complex an applications behaviour in terms of its data requirements, the harder it is to predict accurately. Ultimately, no prediction mechanism is superior in all applications.

Chapter 5

Approaches in Evaluating Prefetching

As identified in chapter 2, the goal of prefetching is to reduce the total execution time of an application program by tolerating the latency of reading data. It is not surprising then, that many designers of prefetching schemes employ evaluation methods based on direct performance measurement of their operation on a running target system. This is done to reflect the ability of their prefetching schemes to reduce the execution time of their applications.

This thesis supports the separate treatment of the prediction and fetching mechanisms involved in prefetching schemes to reflect the efforts of others in porting prediction mechanisms to alternative environments [GK94a, MDK96]. Comparison of these studies in relation to the original studies which proposed a particular prediction mechanism have shown that the results of direct performance measurements made in the context of a particular data set, fetching mechanism, operating system, or hardware configuration do not hold for different application areas.

This chapter examines the reasons for this, and in so doing highlights the deficiencies in the evaluation methods and metrics used in the literature to date. The chapter then proposes an approach to evaluation which enables potential adopters of a prediction mechanism to assess whether a particular prediction mechanism is likely to be suitable to their application area. While work has been

done in establishing evaluation criteria for prefetching which addresses the effect of fetching mechanisms [CFKL95], to date, the evaluation of prediction mechanisms in a context independent manner has been neglected. Accordingly, this chapter focuses on the evaluation of prediction mechanisms.

5.1 The Need for Detailed Evaluation

Despite the recognition of the degree of independence between prediction and fetching mechanisms which studies such as [LM96, MLG92, GK94a] would suggest, prefetching schemes have continued to be evaluated as a whole unit working to reduce execution time for a particular application on a particular hardware / operating system platform. Further to this, the results obtained through such evaluations have been used to imply similar performance improvements for at least the same application area eg. throughout the OODBMS domain. As will be shown in this chapter, this is an invalid premise and the nature of evaluation in this context needs to be examined in more detail.

5.1.1 Operational Parameters

The performance of a prefetching scheme in terms of the reduction in execution time between prefetching and conventional systems is highly sensitive to a large number of factors henceforth referred to as *operational parameters*. These include (but are not limited to) the characteristics of the application, the organisation of data, the characteristics of the operating system, machine resources and contention for resources.

Most of the difficulties in obtaining generally applicable measurements and understanding of prefetching schemes occur as a result of the aforementioned operational parameters, each of which has an impact on the scope of optimisation possible through prefetching.

Despite this, many previous research projects in developing prefetching schemes have evaluated their work on the strength of experimental results obtained by running the prefetching scheme with

a single or limited set of operational parameters. In many cases, this meant running measurement experiments with only a small selection of benchmark applications upon one type of operating system, with a single type of microprocessor, size of memory, and type of disk. By not scientifically projecting the performance of prefetching schemes, no information is generated which shows how the experimentally demonstrated benefits of the prefetching scheme would translate to different sets of operational parameters. This presents a problem for researchers developing prefetching schemes which are expected to be portable to other applications, loads, operating systems, microprocessors, memory configurations and disks.

While it has been shown that certain operational parameters can result in performance degradations [GK94a, MLG92], the current body of published research reveals no generally applicable rules born out of experimentation which would identify those parameters for different prefetching schemes.

5.1.2 Separate Evaluation of Prediction Mechanisms

As a component of prefetching schemes, the performance of a prediction mechanism is also affected by operational parameters. However, the performance results appearing in papers [BKW94, Bes95, GK94a, GA94, GAN93, KGM91, Kna99, KE90, LD92, LM96, MK94, MLG92, PZ91, PGG+95, Smi78] which introduce prediction mechanisms imply that similar performance improvements will hold for broadly similar sets of operational parameters.

A study conducted by Gerlhof [GK94a] showed that differences between between OODB implementations can have a profound effect on the performance of prediction mechanisms. An example of this was found to be the effect of logical or physical persistent identifiers (PIDS). Under applications which exhibited a high degree of locality, the total execution time was found to increase by up to 50% compared to the non-prefetching system. This was attributed to the additional expense

of address mapping which was needed to be performed for every prefetched object. It cannot therefore be presumed that performance results are portable even between different systems in the same application area.

Porting a prediction mechanism to a different fetching mechanism, either in terms of design (indirect or explicit), or in terms of implementation will also yield different performance results from those published in the original paper which presented the prediction mechanism.

If prediction mechanisms are to be ported to other applications or used with different fetching mechanisms, then evaluation of prefetching schemes must at least produce results which capture the qualities specific to the prediction mechanisms themselves.

5.2 Approaches to Evaluation of Prediction Mechanisms

There are a number of evaluation methods for prefetching schemes. This section introduces direct measurement, simulation and mathematical models and discusses their relative advantages and disadvantages in evaluating prediction mechanisms. All of the prefetching schemes present in the literature have been evaluated using direct measurement on real machines. However, there are alternatives to this approach. These are shown in figure 5.1.

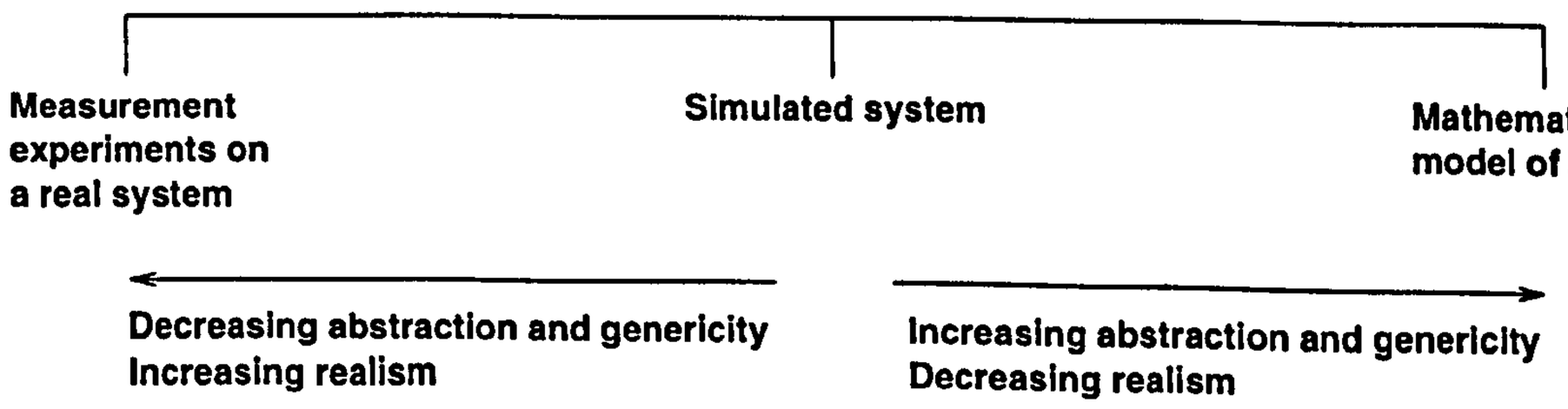

Figure 5.1: The Spectrum of Evaluation Methods

Shown at the extreme left of the spectrum is real system experimentation. With this method,

measurements of some aspect(s) of a real system's behaviour are taken in order to characterise it. At the opposite end of the spectrum, mathematical modelling uses formulae to describe system behaviour. The terms of the formulae may be used to represent, at an abstract level, the components of the system, or behaviour of those components. Simulation lies somewhere between these two extremes. The advantages and disadvantages of each approach are presented in further detail in the following subsections.

In terms of the spectrum of these approaches, simulation can be see as sitting between direct measurement on a real machine and mathematical modelling. Moving towards measurements on real machines, we find ourselves with increasing realism in terms of the analysis possible, and in moving towards mathematical modelling, we find ourselves using increasing abstraction [Sch90]. Thus it seems that there is a tradeoff between the accuracy of analysis and the flexibility and genericity (and ultimately the usefulness) of the analysis.

Simulations all seek to mimic the behaviour of a target entity (in our case a prediction mechanism) by describing the logical relationships between the elements of the target entity which affect situations of interest. Such situations of interest in our case might most obviously include the time taken to run a given application.

5.2.1 Real System Experimentation

Experiments which collect direct measurements of a real system running a prediction mechanism have the advantage that the resulting analysis reflects the actual system. The same cannot be said of either simulation or mathematical modelling in which analysis of the target system is made with the use of abstract facsimiles of the target system.

Since the target system may be influenced by operational parameters outside the control of the measurement experiment such as contention for machine resources, it may be necessary to repeat the

experiment many times in order to produce a sample set of results from which statistically significant results can be obtained. This would allow for noise in the measurements caused by anomalies like external processes on the target machine.

The disadvantage of this approach is that any analysis can only be said to reflect the behaviour of that one target system. It lacks any genericity, and the results of the analysis cannot be said to apply to other similar target systems. Even considering the same target system, the behaviour observed when running a given set of input data cannot be said to be representative of the behaviour under other sets of input data.

Perhaps chief among the disadvantages of this approach to analysis is that is doesn't promote any understanding of the system's behaviour. Instead, it merely presents a disjoint set of behaviour characteristics for a given set of input values. Crucially, this approach doesn't provide any benefit in understanding the behaviour of the system in order to improve upon it.

Direct Measurement of Reduction in Execution Time

The goal of a prefetching scheme is to reduce the execution time of an application by tolerating latency. This leads to the use of reduction in execution time as a metric for showing the effectiveness of a prefetching scheme. However, due to the influence of operational parameters upon the performance of prefetching, this metric is not portable to other contexts. In particular, this technique obscures the performance of the prediction mechanism itself. Instead, it captures a snapshot of the performance of the whole prefetching scheme.

Direct Measurement of Cache Behaviour

Gauging the performance of a prediction mechanism on the basis of the effect it has upon the cache is an approach which is also flawed if performance results are to be portable to other contexts.

Metrics such as reduction in the number of cache misses [GK94a] appear to offer a method of evaluating which is not as dependent upon operational parameters as direct measurement of reduction in execution time. However, the behaviour of the cache will depend upon the management strategy, the size of the cache, and the working set [Den80] set of the application. As such, direct measurement of cache behaviour cannot capture the qualities of the prediction mechanism itself beyond the context of the cache.

For example, consider a mechanism using reduction in cache misses as a metric for evaluating a prediction mechanism. It may be that the prediction mechanism is able to deliver 100% accuracy, but since the size of prediction lookahead is smaller than the latency to be tolerated, the reduction in the number of cache misses is unchanged compared to the non-prefetching case.

Essentially, this approach is flawed, because despite the more abstract nature of the metric, evaluation is still performed within the context of a particular environment.

5.2.2 Mathematical Modelling

Mathematical models of prediction mechanisms would comprise of formulae containing expressions which may be evaluated to produce some answer describing the state of the system. The expressions contain variables and sub-expressions which model the relationships between system inputs and the behaviour of a component (or indeed the whole system).

It is not necessary for the target prediction mechanism to exist, other than to enable validation of the model. Provided that the model accurately represents the behaviour of the target system, it is possible to substitute values (or other subexpressions which model other components) into the model, in order to see how this affects the behaviour of the system.

The main advantage of this technique is that algebraic manipulation of the model's formulae can yield answers to questions such as, "What conditions must be satisfied for this system to behave

optimally?" While it is true that experimentation with a real system might have allowed the optimal behaviour to become apparent, it would undoubtedly would have taken far longer to perform the experiments to do so.

Unfortunately, it may be difficult or impossible to develop a mathematical model which accurately represents the behaviour of the target system [Den80]. In particular, the "discrete event" behaviour exhibited by the target system makes it difficult to model using well understood, continuous functions. For the sake of a tractable model, it is often necessary to model the behaviour of simpler distributions than are actually exhibited by the target system. For example, in the case of mathematical models of prediction mechanisms, one might need to use distributions to model locality of reference in an application as a curve modelling the probability that the next reference will be cache resident.

5.2.3 Simulation

Experimentation on real systems can be prohibitively time consuming or provide results without the information to allow those results to transfer to other inputs, or similar systems. Mathematical modelling relies upon creation of a formulae which can accurately represent the relationships between input and behaviour of the system. Simulation provides an alternative analysis to these two very different techniques.

Simulation [Sch90, DS99] involves experimentation with a facsimile of the real system. This facsimile faithfully mimics the processes of the target system in order to produce similar behaviour without the time expense of experimenting on the real system. Since the simulation is an artificial representation of the target system, the effect of processes external to the simulation can be eliminated.

Previous work by Darmont [DS99] in developing simulations of OODBMSs permits a more

sophisticated method of evaluating prediction mechanisms than is possible through direct measurement. By providing a configurable simulation environment in which the policies for cache management can be set as well as response times for disks and other hardware components, it is possible to obtain performance results for a range of different operational parameters. However, although simulation environments permit performance evaluations to be made for a range of different values of microprocessor speeds, disk speeds and cache models, the results still do not reflect the qualities of the prediction mechanism itself.

5.3 Obtaining Generic Metrics

There is no significant problem in finding the units for generic metrics of prediction accuracy. It suffices to measure the percentage of correct predictions made by the prediction mechanism. However, there is a significant issue of how one measures prediction accuracy. Different prediction mechanisms work in different ways and are unlikely to give a constant level of prediction accuracy. Consider the task of measuring the prediction accuracy of a training-based mechanism based on a lossy pattern memory as seen in [PZ91]. Here, the prediction accuracy will vary depending upon the similarity of recent, distinct data access patterns.

Similarly, prediction coverage can be expressed universally for all prediction mechanisms as the percentage of an applications data accesses for which predictions could be made. Once again, the difficulty is in knowing the correct situations in which to apply the metric so as to give a representative view of the prediction mechanism.

With prediction lookahead, there is also the need to show the cases in which prediction lookahead varies. Additionally, there is a difficulty in finding a generic metric for prediction lookahead. Although prediction lookahead represents the concept of being able to see into the future of an application's data requirements, it cannot be measured in real time, since this would bind the results to

a set of operational parameters. Instead it is necessary to measure logical time. This can be achieved with the unit of prediction. For example, Knafla's static, codified knowledge perspective mechanism predicts page accesses, and so in this case, pages would be the unit of lookahead. Thus, the metric for prediction lookahead is specific to the prediction mechanism.

5.4 Fair Evaluation of Prediction Mechanisms

This section proposes an approach to the evaluation of prediction mechanisms which captures the qualities of the mechanisms themselves rather than their deployment in a particular scenario.

Of course, reduction in execution time remains the most important goal in prefetching. However, as the preceding sections have shown, this is not an appropriate evaluation metric where performance results are intended to be portable to other contexts.

The proposed approach involves acknowledging that although prediction mechanisms have a single goal, they work in such different ways as to make meaningful comparison using a universal set of metrics impossible. The goal of each prediction mechanism is the same: predicting application behaviour in terms of its data requirements. The breadth of strategies used (chapter 4) and the operational parameters which can effect prediction performance (section 5.1.1) are many and varied. The proposed approach abstracts over these complications by addressing the fundamental requirements of prediction mechanisms and designing bespoke micro-benchmarks which highlight the strengths and weaknesses of particular prediction mechanisms with respect to the fundamental requirements of prediction accuracy, prediction lookahead and prediction coverage (section 4.1). In essence, the approach is to treat the micro-benchmarks themselves as the metrics.

5.4.1 Addressing the Fundamental Requirements of Prediction

To evaluate prediction mechanisms in a manner which captures the qualities of a prediction mechanism rather than its use in a particular context, it is necessary to address the best and worst cases for the fundamental requirements of prediction accuracy, prediction lookahead and prediction coverage.

5.4.2 Bespoke Benchmarks

As identified in section 5.3, one of the difficulties in obtaining generic metrics is that prediction mechanisms form their predictions using different information - source code [Mow94], schema [Cha89], object placement [Kna99], stochastic methods [CKV93]. Any attempt to evaluate different prediction mechanisms under a "standard" environment (section 5.2.3) will introduce dependencies specific to that environment which obscure the qualities of the prediction mechanism.

The principle of the approach presented here is therefore to avoid introducing these dependencies and create a set of bespoke benchmarks which attempt to show the strengths and weaknesses of the prediction mechanisms independently.

5.4.3 Evaluation of a First Order Markov Predictor

To demonstrate the approach to evaluation outlined above, this section demonstrates the steps in the creation of bespoke micro-benchmarks to expose the qualities of the First Order Markov (FOM) Predictor presented in [CKV93]. In doing so, this section acts as a guide to the preparation of these benchmarks for any prediction mechanism.

In developing micro-benchmarks for FOM, the following questions must be answered with respect to prediction accuracy, prediction lookahead, and prediction coverage.

- What prediction environment does this mechanism require, and how does the mechanism exploit it?

- What are the metrics used in this mechanism?

- What are the optimal and pathological cases of operation?

Prediction Environment

FOM is a dynamic, tacit knowledge perspective mechanism. The prediction environment which FOM requires supplies the addresses of pages which are accessed as the application runs. Predictions are produced through analysis of the application recently referenced data items (objects or pages) denoted as symbols. The analysis takes place over a fixed sized history window of size n. As the application requests data, predictions are made for the next most probable symbol in the reference stream by constructing a first order Markov chain over the window of references. To achieve this, the most recently referenced symbol is used as a key into the history window and if the key is present at some earlier point in the history, the symbol which most frequently follow the key symbol is chosen as the next predicted symbol.

Metrics Used in the Evaluating the Mechanism

The metrics used in evaluating FOM are as follows. The metric for prediction coverage is the percentage of the application's total number of references for which predictions could be made.

The metric for prediction accuracy is the percentage of the application's predictions which proved to be correct in relation to the symbols actually referenced by the application at run time.

The metric of lookahead is the simply the number of references into the future of the executing application for which predictions can be made.

Microbenchmarks for FOM

The microbenchmarks for FOM are application workloads expressed in terms of the prediction environment which, when processed by the prediction mechanism would exercise its best and worst behavioural aspects. The microbenchmarks for FOM are presented in the form of sequences of references.

The cases for best and worst prediction lookahead are the same in the case of FOM , since it will only ever predict 1 reference into the future of the application by design. As such, a microbenchmark workload to demonstrate prediction lookahead is redundant here and is not presented.

In terms of prediction coverage, FOM can only form prediction when the most recently referenced symbol appears elsewhere in the finite history window. For a history window of size n, we can then say that the sequence of references of the form

$$\lhd i_0, i_1, .., i_{n-1}, i_n \rhd | i_n = i_x \; where \; n \succ x \succeq 0$$

Stated in English, this is simply s sequence of $n-1$ references which includes the symbol at the n^{th} reference.

The sequences corresponding to the best coverage cases can also be presented in terms of an expression in the formal specification language Z.

$$BestCoverage == \{seq : SymbolSequence |$$

$$tail(seq) \_contains\_head(seq) \tag{5.1}$$

$$\}$$

This sequence expression is sufficient to create a set of concrete sequences which represent the cases for best prediction coverage for a first order Markov predictor with a given window size

n. Indeed, in the following chapter these expressions are used to create sets of concrete reference sequences which are then applied to evaluate FOM in the context of a specific application.

The worst case for prediction coverage in FOM is demonstrated by the cases where the most recently referenced symbol is not present at any other point in the history window. In general terms, this is described by sequence of references of the form

$$< i_0, i_1, .., i_{n-1}, i_n \succ | i_n \neq i_x \; where \; n \succ x \geq 0$$

As a set expression from which concrete sequences may be produced, we have the following microbenchmark specified in Z.

$$WorstCoverage == \{ |seq : SymbolSequence| \; /(tail(seq) \_contains\_ head(seq))| \} \tag{5.2}$$

In FOM, as in any other prediction mechanism, a prediction can only be made if the predicated coverage condition has been met. Accordingly, the microbenchmarks which represent the best and worst cases for prediction accuracy are related to the symbol sequences of the best case prediction coverage microbenchmark presented above.

Assuming the conditions for prediction coverage have been met, the prediction accuracy must be assessed on the basis of a comparison between the predicted symbol and the symbol which was referenced next by the application.

In producing the microbenchmarks for prediction coverage in FOM, it is necessary to consider sequences of n references. Here, since we are interested in showing the cases for best and worst prediction accuracy we must consider sequences of $n + 1$ references where n is the size of the history window. This allows us to find the cases where the $n + 1^{th}$ referenced symbol was the same

as the symbol which FOM would have predicted to follow the n^{th} reference. Given this, we can define the microbenchmark for the best case prediction accuracy in the following terms.

$$
\begin{aligned}
BestAccuracy == \{ & seq : SymbolSequence | \\
& \exists subseq : BestCoverage \bullet \\
& subseq = tail(seq) \wedge \\
& most\_often\_follows(head(subseq)) = head(seq) \\
\}
\end{aligned}
\tag{5.3}
$$

In this way, we use the set of sequences defined by the $BestCoverage$ microbenchmark to act as the basis for the definition for the $BestAccuracy$ benchmark.

The microbenchmark which represents the worst case for prediction accuracy in FOM is given by the following expression.

$$
\begin{aligned}
WorstAccuracy == \{ & seq : SymbolSequence | \\
& \exists subseq : BestCoverage \bullet \\
& subseq = tail(seq) \wedge \\
& most\_often\_follows(head(subseq)) \neq head(seq) \\
\}
\end{aligned}
\tag{5.4}
$$

Note that this case is also build upon the set of symbol sequences identified in $BestCoverage$.

5.4.4 Evaluation of the OSP Prediction Mechanism

The Object Structure Prefetching (OSP) prediction mechanism [Kna99] predicts which pages will be accessed in an OODB. This is achieved by examining the layout of objects on the pages of the

database. For each page, those objects which reference objects on other pages are identified as outward referring objects (*oro*s). The application specific parameter, prefetch object distance (*pod*) is set by the user as the number of objects which could be processed by the application in the time taken for a page fault.

A chain of objects of maximum length dictated by the pod is established "upstream" of the oro and within the same page. The first object in the chain is identified as the prefetch start object (*pso*). The pso has the restriction that it must be located on the same age as the oro. Additionally, where the initial choice of pso has paths from it which lead to different external pages, a child of the initial pso is chosen to be the pso. The pso must have the property that the only external page which can be reached down any path from the pso leads to the page containing the object referenced by the oro. If the oro itself refers to more than one external page, then no pso can be found. At runtime, when the pso is accessed, it a prefetch is issued for the page containing the object referenced by the oro.

Prediction Environment

The prediction environment for OSP is one of object distribution over pages of the database. No consideration is given to the code which manipulates the database or the schema which defines it. As such, the microbenchmarks are expressed in terms of the relationships between objects on pages of the database.

Metrics Used in the Evaluating the Mechanism

The metrics for the evaluation of this prediction mechanism are as follows.

Prediction coverage is the percentage of all oros on a page through which we can identify a pso.

Prediction accuracy is the percentage of psos on a page through which all paths lead to an object on the same page as the oro.

Prediction lookahead is measured as the number of objects between the pso and the oro.

Microbenchmarks for OSP

The microbenchmarks for OSP are page layouts expressed in terms of the prediction environment. Specifically, the best and worst cases of prediction coverage, prediction lookahead and prediction accuracy are presented in terms of sets which represent the relationships between objects located on pages of the database. Pages are represented as sets of objects.

The case of best prediction coverage occurs when we are able to disambiguate a pso. That is, an object which is not an oro and which has a path through an oro such that the only external page which can be reached down this path is the page containing the oro. The object relationships for best prediction coverage is then expressed as the following set of injective mappings from pso to oro objects on a page.

$$
\begin{aligned}
BestCoverage =\!\!- \{ & (pso \mapsto oro) : Object \rightarrowtail Object \mid \\
& \exists p_1 : Page \bullet \\
& \exists referenced : Object \bullet \\
& \exists_1 p_2 : Page \bullet \\
& oro \epsilon p_1 \wedge \\
& pso \epsilon p_1 \wedge \\
& reference d \epsilon p_2 \wedge \\
& pso \neq oro \wedge \\
& p_1 \neq p_2 \wedge \\
& pso\_can\_only\_reach\_p_2 \\
\} & ,
\end{aligned}
\tag{5.5}
$$

The case for worst prediction coverage for OSP is achieved whenever we are unable to determine a pso for an oro, or when an oro refers to objects on more than one external page. Expressed as a set of oros for which we are unable to find a pso, here is the microbenchmark relating to worst prediction coverage in OSP.

$$
\begin{aligned}
WarstCoverage == \{ ora : Object \mid \\
& \exists p_1 : Page \bullet \\
& \exists referenced : Object \bullet \\
& oro \in p_1 \wedge \\
& referenced \notin p_1 \wedge \\
& \nexists pso : Object \bullet \\
& pso \neq oro \wedge \\
& pso\_can\_only\_reach\_p_1 \\
\}
\end{aligned}
\tag{5.6}
$$

In OSP, the prediction lookahead is defined upon the set of object relationships between pso and oro objects which were established for the *BestCloverage* set. The application specific parameter pod is used in establishing a threshold value for acceptable prediction lookahead in terms of the number of objects. The pod is set by the user of the application in response to the characteristics of the application and database. Our interest is in measuring how many of the set of mappings between pso and oro established in the *BestCloverage* set have a shortest path between pso and oro equal to the pod. In these cases, the lookahead has reached its maximum value possible value. If the pod has been established appropriately by the user, we can expect the latency of the page faults to be fully tolerated.

$$BestLookahead == \{lookahead\_achieved : BestCoverage \mid$$

$$\exists pod : \mathbb{N} \bullet$$

$$shortest\_path\_betweenlookahead\_achieved = pod$$

$$\}$$

(5.7)

$$WorstLookahead == \{insufficient\_lookahead : BestCoverage \mid$$

$$\exists pod : \mathbb{N} \bullet$$

$$shortest\_path\_betweenlookahead\_achieved \neq pod$$

$$\}$$

(5.8)

In OSP, predictions are made over which page will be accessed next during a traversal of the object structure. In this environment, the best and worst cases for prediction lookahead are the same since the predictions will always be correct due to the nature of the analysis: the pso is always chosen such that there is only one possible next page. As such, microbenchmarks are redundant in the case of prediction accuracy for OSP.

5.4.5 Advantages and Disadvantages of Approach

This approach to evaluation captures the qualities of the prediction mechanism itself rather than its application to a set of operational parameters. It highlights the areas in which a particular mechanism is strong or weak, thereby empowering potential adopters to make informed choices on whether it is appropriate for them.

In creating the micro-benchmarks, there is a reliance upon an understanding of the mechanism, its operation, and how it is affected by relevant characteristics of the application. This may only be feasible for the designer of a prediction mechanism to create if the operation of the mechanism is sufficiently complex.

For users of the micro-benchmarks, interpretation of the results relies upon them understanding the micro-benchmark and being able to relate it to their application. This clearly involves more effort than interpreting a simple numerical result from the execution of a "standard" macro-benchmark.

5.5 Summary and Conclusions

This chapter discussed the need for separate evaluation if prediction mechanisms are to be ported to other fetching mechanisms and applications. An analysis of possible evaluation methods was made which concluded that all existing evaluation methods evaluate a prediction mechanism in the context of a particular set of operational parameters.

An approach to evaluation was presented which captures the qualities of the prediction mechanisms rather than their performance in the context of a particular set of operational parameters. The process performing such an evaluation was then outlined using the evaluation of both a First Order Markov predictor and the Object Structure Prefetching predictor. The framework for evaluation as presented in this chapter allows for the qualitative evaluation of a mechanism's fitness for a particular application by having the potential adopter of a prediction mechanism visually inspect the microbenchmarks and make a judgement on whether the cases represented by them are generally representative of the application. Alternatively, the potential adopter of the prediction mechanism can perform pattern matching of the microbenchmarks over the target application and in doing so arrive at a quantitative evaluation of a prediction mechanism in relation to its deployment in a specific application. An example of the quantitative evaluation of a prediction mechanism using the

microbenchmarks is demonstrated in the following chapter.

Chapter 6

Demonstration of the Evaluation Framework

The previous chapter presents a method of evaluating prediction mechanisms in a way which allows their efficacy to be assessed without the necessity of implementing the prediction mechanism. The evaluation framework also promotes the evaluation of prediction mechanisms in such a way as to abstract away from the effects of a prediction mechanism's implementation in terms of the operational parameters from the machine.

The purpose of this chapter is to demonstrate the process of taking a set of microbenchmarks for a prediction mechanism and evaluating the suitability of the mechanism to a particular application. Specifically, this chapter presents the evaluation of the FOM predictor with a history window of length 6 in the context of the OO7 application using the microbenchmarks presented in the previous chapter.

Prior to conducting the evaluation, consideration is given to the roles and responsibilities of the prediction mechanism designer and potential adopter of the prediction mechanism in performing such a quantitative evaluation. This is presented by way of providing a roadmap of the tasks to be

performed and indicate the degree of effort involved by the designer of the prediction mechanism in producing the microbenchmarks and the potential adopter in assessing the degree of correspondence between the microbenchmarks and the target application.

The application area is discussed to give details of the application and what has been collected from it to allow comparisons with the microbenchmarks.

Finally, the FOM microbenchmarks are shown being applied to the OO7 application and a subsequent verification of the results is given using a concrete implementation of the FOM predictor.

6.1 Roles and Responsibilities

The evaluation framework presented in the preceding chapter asks more of the designer of a prediction mechanism than simply running the mechanism over a sample application and publishing the reduction in execution time. Significant effort is involved on the part of the designer (or someone else who knows how the mechanism operates) to produce microbenchmarks which demonstrate the best and worst cases of operation.

In a similar vein, the process of taking the microbenchmarks and pattern matching them to the target application in order to determine the degree of correspondence of the application to the microbenchmarks also involves significant effort.

6.1.1 Roadmap for evaluation of FOM in OO7

The tasks necessary to perform a quantitative evaluation of FOM in the context of the OO7 benchmark are provided below. The first two tasks relate to the creation of the microbenchmarks and would be performed by the designer of a prediction mechanism as part of publishing results of the mechanism. In the case of FOM, these have been presented in the preceding chapter.

The last three tasks are to be performed by the potential adopter of the prediction mechanism.

These tasks are the focus for this chapter.

1. Analyse the operation of the prediction mechanism and produce microbenchmarks to highlight the best and worst cases for accuracy, lookahead and coverage.

2. Codify the best and worst cases in terms of a set of sequence expressions to capture concisely capture all possible sequences which match the best and worst criteria.

3. Capture the application's behaviour in the same terms as the microbenchmark's OO7 object trace.

4. Use each microbenchmark to generate a corresponding set of concrete sequences of references. Match the concrete patterns to the reference trace produced by OO7.

5. Assess the degree of correspondence between the sets of concrete sequences representing the different microbenchmarks and the OO7 trace.

6.2 Capturing the Application's Behaviour

The application against which FOM is to be evaluated is the t1 small db traversal program from the OO7 benchmark running on the Pjama orthogonally persistent system.

The FOM predictor operates in a prediction environment in which only the stream of data items requested by the application is available to the predictor. Accordingly, the microbenchmarks for FOM presented in the previous chapter are stated in terms of sequences of references to symbols.

To establish the degree of correspondence between the microbenchmarks and the target application, it is necessary for the target application to be expressed in the same terms as the microbenchmarks. To achieve this, the Pjama runtime system was instrumented to record the object identifiers of all objects referenced as the OO7 application ran. The resulting object trace was recorded to a file

and used as the basis for the quantitative evaluation.

Although expressed as a stream of references to objects, the OO7 object trace is not yet in the same form as the microbenchmarks, since they are expressed in terms of the reference window of size n. In order for pattern matching between the microbenchmarks and the application to occur, a simple Java program is written to produce a sequence of the 7 most recently referenced symbols. These will be referred to in the following evaluation as the OO7 trace sequences.

Referened symbol	output trace sequence
A	A
B	A,B
A	A,B,A
C	A,B,A,C
D	A,B,A,C,D
C	A,B,A,C,D,C
A	A,B,A,C,D,C,A
E	B,A,C,D,C,A,E
F	A,C,D,C,A,E,F

6.3 Tools Employed in the Evaluation

The evaluation which follows makes extensive use of a relational database to represent the sequences of references used in the evaluation. A relational database was chosen for its efficiency in dealing with set oriented processing of data. However, due to the expressive limitations of SQL compared to a language such as Transact-SQL, Java has been employed to perform operations such as pivoting the stream of OO7 object references into the OO7 trace sequences depicted above.

6.4 Making Concrete Sequences from Microbenchmarks

Examination of the FOM microbenchmarks for best prediction coverage and best prediction accuracy (given in the preceding chapter) shows that the set of reference sequences which constitute best prediction accuracy are based on the set of sequences for prediction coverage.

Examining the microbenchmark for best prediction coverage, it can be seen that it specifies all sequences of references which meet the following criteria: a sequence of n referenced symbols where the n^{th} referenced symbol occurs at least once in the preceding $n-1$ references.

Examining the microbenchmark for best prediction accuracy, it can be seen that, given a history window of size n, the sequences are all in terms of $n+1$ references. This is because we define the accuracy of the prediction mechanism on the basis of the symbol which follows the n recently referenced symbols appearing in the history window.

The analysis presented here is for a FOM predictor with a history window of size $n = 6$. The reference sequences are modelled as records in a relational table (ref1, ref2,..) such that the field ref1 indicates the reference in the sequence which immediately preceded the reference stored in ref2.

Any reference sequence of length 6 where the 6th referenced symbol occurs at least once in the preceding 5 references can consist of references to a maximum of 5 distinct symbols. The concrete sequences for the best prediction coverage could therefore be expressed by forming the Cartesian product of 5 symbols over the 6 places in the sequence and applying the predicate from the microbenchmark that the 6th referenced symbol must match one of the first 5 references in the sequence.

The sequences corresponding to best prediction accuracy are 7 references long and have the first 6 referenced symbols matching the 6 references of the concrete sequences for best prediction coverage. Accordingly, if we define the reference sequences for best prediction coverage in terms of 7 references instead of the required 6, we can efficiently create the set of concrete sequences for prediction accuracy by producing a subset of the best coverage sequences.

To do this, we define two tables

```
6REFS( int id, char value )
7REFS( int id, char value )
```

where 6REFS is populated with the values

```
1, A
2, B
3, C
4, D
5, E
6, F
```

And 7REFS is populated with the values

```
1, A
2, B
3, C
4, D
5, E
6, F
7, G
```

We now define the table to hold all the concrete sequences of references corresponding to the BEST_COVERAGE as

```
BEST_COVERAGE(
char ref1,
char ref2,
char ref3,
char ref4,
char ref5,
char ref6,
char ref7
)
```

BEST_COVERAGE is then populated in accordance with the predicates of the microbenchmarks using the data from the 6REFS and 7REFS tables using the following SQL.

```
INSERT INTO BEST_COVERAGE
SELECT r1.value AS r1,
r2.value AS r2,
r3.value AS r3,
r4.value AS r4,
r5.value AS r5,
r6.value AS r6,
r7.value AS r7
FROM 6REFS AS r1,
6REFS AS r2,
6REFS AS r3,
6REFS AS r4,
```

```
6REFS AS r5,
6REFS AS r6,
7REFS AS r7
WHERE r6.value=r1.value
Or r6.value=r2.value
Or r6.value=r3.value
Or r6.value=r4.value
Or r6.value=r5.value;
```

This captures all concrete sequences which for which references ref1..ref6 in each sequence match the predicate defined by the microbenchmark for best prediction coverage. The 7th reference is unbound by the predicate and so contains both cases where the referenced symbol has been encountered before in the reference window and where the referenced symbol is unprecedented in the reference window.

The concrete sequences in BEST_COVERAGE are stated in terms of the symbols appearing in the 6REFS and 7REFS tables. There are two issues to solve.

Firstly, since we are interested in matching the patterns of reference sequences from BEST_COVERAGE to the OO7 trace sequences, we need to relate the two sets of symbols used. The symbols used in the OO7 trace sequences are of the form:

```
java.lang.Thread@EE300130/EE3334A0,
java.lang.Thread@EE300130/EE3334A0,
java.util.HashtableEntry@EE300200/EE333D88,
oo7.OO7@EDC29F10/EDB80280,
..
```

The reference sequences in the BEST_COVERAGE table are of the form:

```
A,A,B,D,B,D,E
```

We require some method of normalising the two sources of reference sequences so that they may be compared.

Secondly, in terms of the concrete sequences in BEST_COVERAGE, there are records which we should consider as duplicates for the purposes of pattern matching between the microbenchmark and OO7. For example, we wish to consider the following records as duplicates and eliminate one of them to produce a set of distinct set of derived patterns over the concrete sequences.

```
A,A,B,D,B,D,E
C,C,A,E,A,E,B
```

To address both of these issues, a Java application is built to derive patterns from a sequence of references such that the first unique referenced symbol in a sequence is labelled 0, the second unique referenced symbol is labelled 1, etc.

In this way, the concrete sequences presented above are translated to two identical patterns

```
0,0,1,2,1,2,3
0,0,1,2,1,2,3
```

We can now read the patterns back into the database and perform a select distinct operation to eliminate duplicate patterns.

This same pattern derivation program can be applied to the OO7 trace sequences to produce patterns of references which can be directly compared with the derived patterns for the microbenchmarks.

The pattern derivation program is run over each record in the BEST_COVERAGE table to produce the BEST_COVERAGE_DERIVED_PATTERNS table

```
BEST_COVERAGE_DERIVED_PATTERNS( int ref1,
int ref2,
int ref3,
int ref4,
int ref5,
int ref6,
int ref7
)
```

The table

```
DISTINCT_BEST_COVERAGE_DERIVED_PATTERNS(
int pattern_id,
int ref1,
int ref2,
int ref3,
int ref4,
int ref5,
int ref6,
int ref7
)
```

is created with a similar schema to the BEST_COVERAGE_DERIVED_PATTERNS table, but which includes a primary key pattern_id field to identify the pattern. An "insert select distinct" query is performed upon the DISTINCT_BEST_COVERAGE_DERIVED_PATTERNS table to insert distinct sequences corresponding the the best coverage cases from BEST_COVERAGE_DERIVED_PATTERNS.

Rather than produce the concrete patterns relating to worst prediction coverage, we recognise that any OO7 trace sequence which cannot be matched to one of the patterns in the DISTINCT_BEST_COVERAGE_DERIVED_PATTERNS conforms to the worst case scenario for prediction coverage represented by the microbenchmark.

To generate the set of reference sequences which correspond to the microbenchmark for best prediction accuracy, it suffices to take the sequences in DISTINCT_BEST_COVERAGE_DERIVED_PATTERNS and select those cases in which the symbol which most frequently followed the symbol at field ref6 in fields ref1..ref5 was present in field ref7. Due limitations in the expressive power of SQL, this step was performed using a simple Java program to perform the filtering.

The results were used to populate the table

```
DISTINCT_BEST_ACCURACY_DERIVED_PATTERNS(
int pattern_id,
int ref1,
int ref2,
int ref3,
int ref4,
int ref5,
int ref6,
int ref7
)
```

The set of patterns which correspond to the predicates of the microbenchmark for worst prediction accuracy are obtained by creating the table

```
DISTINCT_WORST_ACCURACY_DERIVED_PATTERNS(
int pattern_id,
int ref1,
int ref2,
```

```
int ref3,
int ref4,
int ref5,
int ref6,
int ref7
)
```

and inserting into it all records from the

DISTINCT_BEST_COVERAGE_DERIVED_PATTERNS

table which do not occur in the

DISTINCT_BEST_ACCURACY_DERIVED_PATTERNS table.

Recall that microbenchmarks for prediction lookahead are redundant here, since the best case scenario is the same as the worst case scenario. That is, assuming a prediction can be made, it can only be made for the next reference, rather than the next n references into the future.

6.5 Assessing the Degree of Correspondence

Now that we have the tables DISTINCT_BEST_COVERAGE_DERIVED_PATTERNS,

DISTINCT_BEST_ACCURACY_DERIVED_PATTERNS, and

DISTINCT_WORST_ACCURACY_DERIVED_PATTERNS to represent the interesting cases of our microbenchmarks, we can assess the degree of correspondence between the microbenchmarks and the OO7 application.

Taking the OO7 trace sequences, we create and populate the table

```
OO7_DERIVED_PATTERNS(
int ref1,
int ref2,
int ref3,
int ref4,
int ref5,
int ref6,
int ref7
)
```

by running the pattern derivation Java program over each of the OO7 trace sequences.

By performing a "select count(*)" query over the OO7_DERIVED_PATTERNS table, we find that the total number of sequences is 1179038. In an implementation of FOM, the predictor would be asked to make a prediction for each of these sequences. The degree of overlap in terms of the number of records from each of the tables representing microbenchmarks whose ref1..ref7 fields match the ref1..ref7 fields of OO7_DERIVED_PATTERNS indicates the degree of overlap for that case.

To assess the degree of correspondence between the microbenchmark representing best prediction coverage, we form the following join query.

```
SELECT oo7dp.ref1,
oo7dp.ref2,
oo7dp.ref3,
oo7dp.ref4,
oo7dp.ref5,
oo7dp.ref6,
oo7dp.ref7
FROM
DISTINCT_BEST_COVERAGE_DERIVED_PATTERNS AS bcdp,
OO7_DERIVED_PATTERNS AS oo7dp
WHERE (((bcdp.ref1)=[oo7dp].[ref1]) AND
((bcdp.ref2)=[oo7dp].[ref2]) AND
((bcdp.ref3)=[oo7dp].[ref3]) AND
((bcdp.ref4)=[oo7dp].[ref4]) AND
((bcdp.ref5)=[oo7dp].[ref5]) AND
((bcdp.ref6)=[oo7dp].[ref6]) AND
((bcdp.ref7)=[oo7dp].[ref7]));
```

Counting the rows of this result set show that there is a match of 941021 records from the 1179038 records present in OO7_DERIVED_PATTERNS. So there 78% of the time, FOM would be able to make predictions if it were deployed in the scenario under analysis.

Simple subtraction shows that this leaves 238017 which correspond to the case of worst prediction coverage. 22% of the time, FOM would be unable to make a prediction if it were to be used here.

Performing the same join query with the DISTINCT_BEST_ACCURACY_DERIVED_PATTERNS

table, we see that there is a correspondence of 247664 records. That is, of the 1179038 records in OO7_DERIVED_PATTERNS, we would expect FOM to be able to form a prediction and that the prediction would be correct in 247664 of the records. This equates to 21% of all references made by the application.

Performing the join query with the DISTINCT_WORST_ACCURACY_DERIVED_PATTERNS table, we see that there is a correspondence of 695537 records. That is, of the 1179038 records in OO7_DERIVED_PATTERNS, we would expect FOM to be able to form a prediction and that the prediction would be incorrect in 695537 of the records. This equates to 59% of all references made by the application.

6.6 Verification of Results

The evaluation presented above has been derived using the microbenchmarks presented in the preceding chapter. As stated in that chapter, the purpose of the framework for evaluation presented in this thesis is to promote the evaluation of prediction mechanisms in a way which is independent of their use in a particular application or binding to a particular set of operation parameters. However, this chapter recognises the importance of being able to apply the microbenchmarks to a particular application and gather quantitative results on the performance of the predictor in the context of their application were they to build it.

Naturally, the only way the validate the results achieved using this approach is to construct the FOM prediction mechanism and run it with the same sets of inputs to verify that the results obtained in the quantitative evaluation of the mechanism are realistic.

A Java implementation of the FOM prediction mechanism was constructed which exported the following interface.

```
Constructor Summary:
```

```
FOM(int windowSize, int predictions)

Method Sumary:
Void addReference(java.lang.Object reference)
java.util.List predict()
```

This interface allowed the construction of a FOM predictor with a specified window size which would return a ranked list of next most probable objects, given that the size of the list is determined by the predictions parameter.

The addReference method allows an external caller to pass a reference to the predictor on the understanding that it will be used to predict the next most likely object when the predict method is called.

A wrapper program was constructed to read in the object references from the OO7 trace file. For each object reference in the trace file, a call is made to the instance of the FOM predictor to invoke the addReference method passing the object read from the OO7 trace file. Immediately after the call to addReference, the predict method is invoked to obtain the next most likely object. In the case of no prediction coverage, a list with no elements is returned. The wrapper program keeps track of the number of times for which predictions can and cannot be made and the cases in which a prediction is made which proves to be correct or incorrect.

Upon running the program it is noticed that there is a slight discrepancy between the figures quoted by the FOM wrapper program and those obtained through the application of the microbenchmarks. Specifically, the number of opportunities to perform prediction (the number of references) was established as 1179044 in the FOM wrapper program compared to 1179038. This can be attributed to the fact that the microbenchmarks analysis was performed by taking successive sets of seven references from the OO7 object trace. 1179044 is not evenly divisible by 7 and so there are 6 references at the end of the application trace for which no analysis was performed. This complication aside, the results from running the FOM wrapper over the OO7 objects trace produced the same

percentages for prediction coverage and accuracy as the quantitative evaluation presented above.

6.7 Conclusion

This chapter demonstrated the utility of the evaluation framework presented in the preceding chapter by showing how they can be applied to a specific setting to give a quantitative evaluation of the suitability of a prediction mechanism to an application. This evaluation is independent of machine loading of specification. One of the attractions of such an approach is that judgements can be made about the suitability of a prediction mechanism to a particular application without the necessity of building the prediction mechanism and running the application over it. In order to verify the approach taken here, an implementation of the FOM prediction mechanism was created and run against a OO7 application trace to ensure that the observed prediction coverage and accuracy reflected the results of the microbenchmark quantitative evaluation.

Chapter 7

The Sympa Prediction Mechanism

The classification and taxonomy of prediction mechanisms presented in chapter 4 has provided the motivation to create a prediction mechanism which combines elements from different areas of the classification in a way which places it in a previously unoccupied area of the taxonomy.

This chapter proposes the Sympa prediction mechanism as a novel approach to prediction in persistent, object oriented environments in which both application code and data are available to the prediction environment. Sympa can be related to (and seen as an extension of) previous prediction mechanisms targeted at object oriented databases. Additionally, it utilises the concepts of inter-procedural optimisation[Hal91] to promote greater prediction lookahead.

Prior to the conceptual discussion of Sympa, the features of the application area and associated prediction environment in which Sympa was designed to work are explained. This includes a discussion of object oriented concepts and persistent object systems.

Chapter 5 presents an approach to evaluating prediction mechanisms in a way which captures the qualities of the prediction mechanisms rather than their use in a particular context. In addition to extending previous work in prediction, this mechanism serves as a target for the evaluation using the approach of chapter 5. The evaluation of Sympa is discussed in chapter 8.

7.1 Object Orientation

Sympa is targeted at object oriented persistent systems. For ease of exposition, this chapter establishes the concepts of object orientation and persistence as they relate to Sympa. In doing so, the prediction environment and characterising features of the applications are defined.

The object oriented paradigm has gained favour as the development model of choice for most new application programs [SC97]. The approach is based on an intuitive correspondence between a software simulation of a physical system and the physical system itself. This simulation is modelled using the key concepts of objects, classes and methods.

7.1.1 Assumptions and Terminology

The object oriented paradigm has been adopted by a number of programming languages and database systems. However, between the implementations of the paradigm, there are differences in the facilities provided. These differences complicate the discourse of this chapter. For this reason the characteristics of the target object oriented system used by Sympa are assumed to be similar to those of the Java programming language [JSGB00].

Briefly then, here are some of the conventions assumed in this chapter.

- **Objects.** The term "object" will be used in the generic sense to refer to both classes as well as entities instantiated from classes (class instances).

- **Members.** Fields and methods. Associated visibility modifiers.

7.1.2 Object Persistence

With the continuing expansion of computer systems in areas such as office automation, industrial CAD/CAM, and CASE tools, there is a growing need for systems which can support objects with

7.1 Object Orientation

Sympa is targeted at object oriented persistent systems. For ease of exposition, this chapter establishes the concepts of object orientation and persistence as they relate to Sympa. In doing so, the prediction environment and characterising features of the applications are defined.

The object oriented paradigm has gained favour as the development model of choice for most new application programs [SC97]. The approach is based on an intuitive correspondence between a software simulation of a physical system and the physical system itself. This simulation is modelled using the key concepts of objects, classes and methods.

7.1.1 Assumptions and Terminology

The object oriented paradigm has been adopted by a number of programming languages and database systems. However, between the implementations of the paradigm, there are differences in the facilities provided. These differences complicate the discourse of this chapter. For this reason the characteristics of the target object oriented system used by Sympa are assumed to be similar to those of the Java programming language [JSGB00].

Briefly then, here are some of the conventions assumed in this chapter.

- **Objects.** The term "object" will be used in the generic sense to refer to both classes as well as entities instantiated from classes (class instances).
- **Members.** Fields and methods. Associated visibility modifiers.

7.1.2 Object Persistence

With the continuing expansion of computer systems in areas such as office automation, industrial CAD/CAM, and CASE tools, there is a growing need for systems which can support objects with

a wide range of *life times*. The life time of an object spans from the point of its creation to the point where it is not required by any of the applications which reference it. Since many computer applications model real-world entities which exist for a long time, there is a need for these *persistent applications* to support life times which can outlast not only the execution of an application, but subsequent versions of that application.

An example of the type of long-lived data used by persistent application systems might be data associated with people. Since people are almost certainly expected to have extent beyond the execution of any application modelling them, support must be provided for the long-lived data associated with those people. It would be unreasonable to reconstruct the objects which model the people every time the application was executed. Indeed, in many cases, the reconstruction of the data may only be possible through weeks of laborious keyboard input. This solution is clearly inappropriate for all but the very smallest sets of long-lived data.

Formally, a data value's *persistence* [Atk78] is the period of time for which that data value exists and is usable. The support of persistence therefore requires mechanisms to manage objects for their full life time regardless of how long or short that may be. As a result, persistent systems support both data values which exist only for the duration of the executing application (which may be thought of as transient) and data values which can transcend the executing application that created it, and may even be seen as independent of any one application.

There are a number of ways in which object persistence may be achieved [AM95, SC97]. Sympa addresses prediction in the context of systems which provide access to the class schema and method code of an application program. Such an environment is available in the orthogonally persistent Java implementation, PJama [ADJ+96].

7.1.3 Persistent Object Applications

Persistent object systems are typified by the use of large, complex structures. Analysis of benchmarks for persistent object systems [DPS98, CDN93] which were designed to be representative of typical target application exposes the characterising features of those applications. These benchmark applications are typified by features such as schemas with a large number of classes, with many complex associations between both the classes and the class instances.

7.2 Concept of Sympa

The principal difference which separates Sympa's prediction mechanism from those of extant database prefetching schemes is its use of application code. The central concept is that instead of predicting data access independently of the semantics of the application, it is possible to use the application program's code to better inform the prediction process. This section illustrates how analysis of the classes and methods of an application can be used to expose the data requirements of an application.

7.2.1 Schema and Relationships Between Data

The class schema, the set of classes which form the application, determine how the application's data (in the form of class instances) is structured. The classes define fields to store data associated either with the class itself or with each class instance. The fields may be scalar or reference fields holding atomic values or object values respectively. The fields can also be designated static, in which case the field (and its implied relationship) applies to the class itself. By contrast, instance fields establish relationships which apply to the class instances instantiated from the class. These fields, scalar or reference, static or instance comprise the state of the class and all its class instances.

The UML diagram in figure 7.1 shows an association between two classes A and B. The instance field `A.x` is defined by class A to reference a class instance of B. This can be interpreted as an

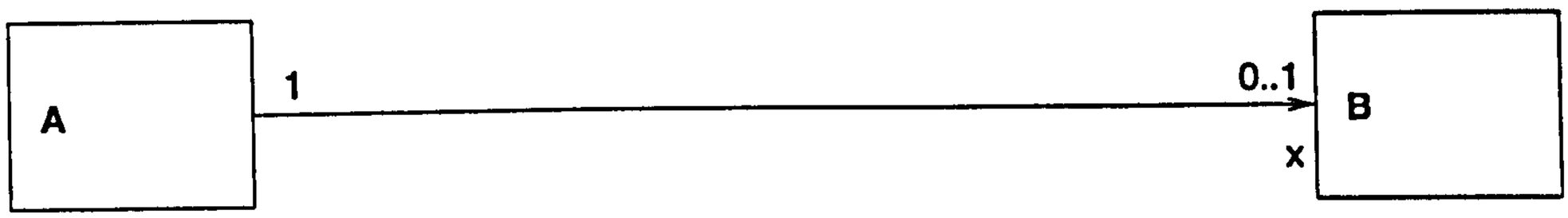

Figure 7.1: UML diagram of the relationship between instances of classes A and B via the x field.

explicit relationship between A and B. In this way, the fields which comprise the state of classes and class instances capture the possible set of relationships between objects in the object graph.

7.2.2 Methods and Navigation of the Object Graph

While the class schema establishes the nature of the relationships or pathways between objects in the object graph, it is the invocation of the application's methods which causes the traversal of relationships between the objects. Methods define application behaviour in terms of which object relationships are used to access or modify the data.

Method analysis reveals which of the relationships established in the class schema may be traversed by a particular method and the order in which they may be traversed.

Program 7.1 The simple() method which causes traversal of the relationships between class instances of the classes A1, B1, and C1.

```
class A1{
    private B1 b;
    private C1 c;

    // Constructor
    public A1(){
    b = new B1();
    c = new C1();
    }

    // Dereference fields
    public void simple(){
    System.out.println( b.x );
    System.out.println( c.y );
    }
}
```

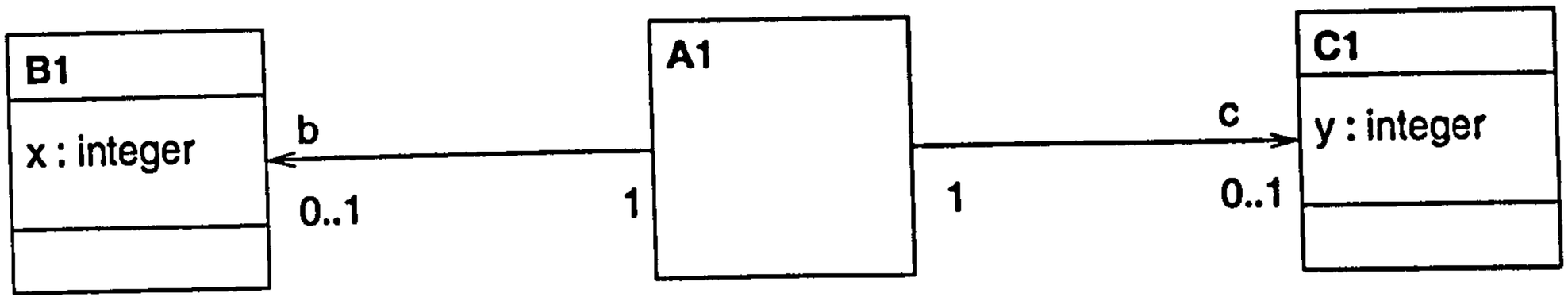

Figure 7.2: UML diagram of the relationships between class instances of classes A1, B1, and C1

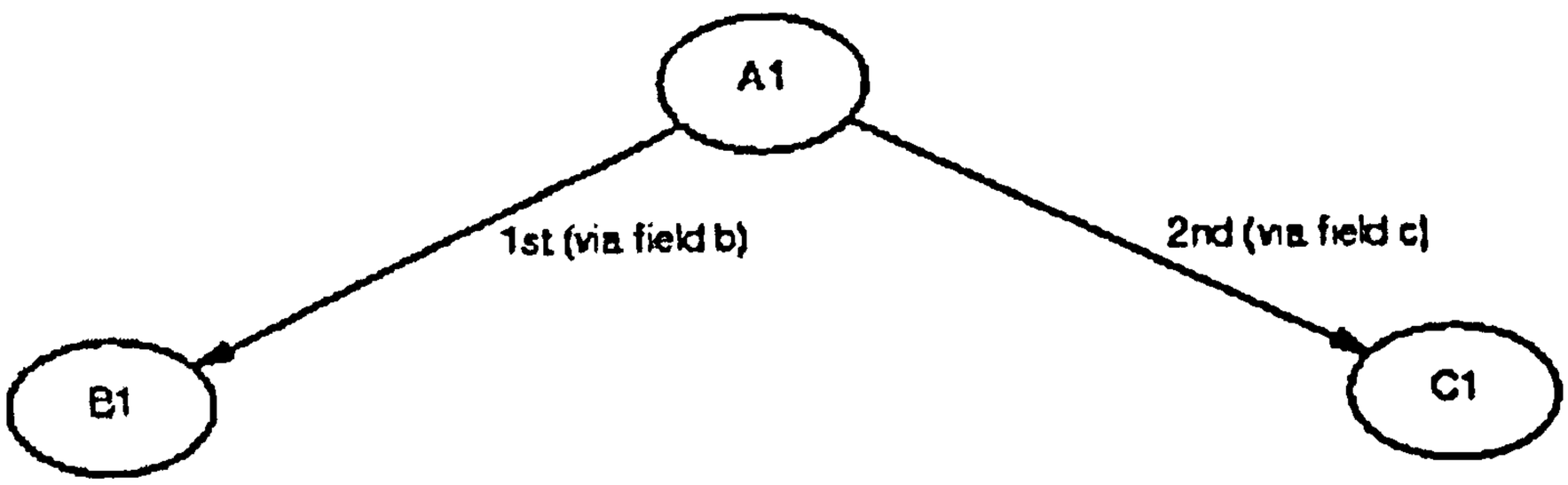

Figure 7.3: Reference Shape for the simple() method

Program 7.1 shows a simple class with two instance fields referencing objects of classes `B1` and `C1` which have a single instance field `x` and `y` respectively. The schema is represented in the UML diagram of figure 7.2. Invocation of the method `simple()` results in the traversal of the relationships from `A1` to `B1` and then `C1` by printing the public integer field `x` from the class instance referenced by `b` and printing the public integer field `y` from the class instance referenced by `c`. Analysis of the method code for `simple()` reveals the order of the relationship traversals caused by the method's execution. The concept of a *reference shape* for an instance method is proposed to show the order in which the relationships defined by the class schema of an application are traversed by a method. The reference shape for the `simple()` method of program 7.1 is depicted in figure 7.3. Each node in the reference shape shown corresponds to a class which defines the class instances. The edges between the nodes show the traversal order of object relationships

specified by the fields of the classes and class instances. Here, the reference shape shows that of the two fields, `b` and `c`, the `b` field is traversed before the `c` field.

The reference shape for `simple()` (figure 7.3) derived in the example above doesn't provide much more information than the original analysis of the schema. In terms of prediction lookahead, only one reference lookahead is possible. However, in cases where method invocations are performed in the course of executing other methods, referencing shapes can be established which are considerably larger than in the above example and can therefore provide much greater prediction lookahead.

Consider the code fragment shown in program 7.2. Here, as in program 7.1, the class has two fields named `b` and `c`. However, in this case, the method `simple2()` will invoke methods defined upon the class instances referenced by `b` and `c` (as shown in figure 7.4 rather than access its public fields. Each of those methods has a reference shape also. By examining chains of method invocations over the program using inter-procedural analysis, it is possible to establish large reference shapes which have the potential to generate long prediction lookaheads. Given the descriptions of classes `B2` and `C2` (program 7.3), the reference shape for the invocation of the `simple2()` method and its descendents is shown in figure 7.5.

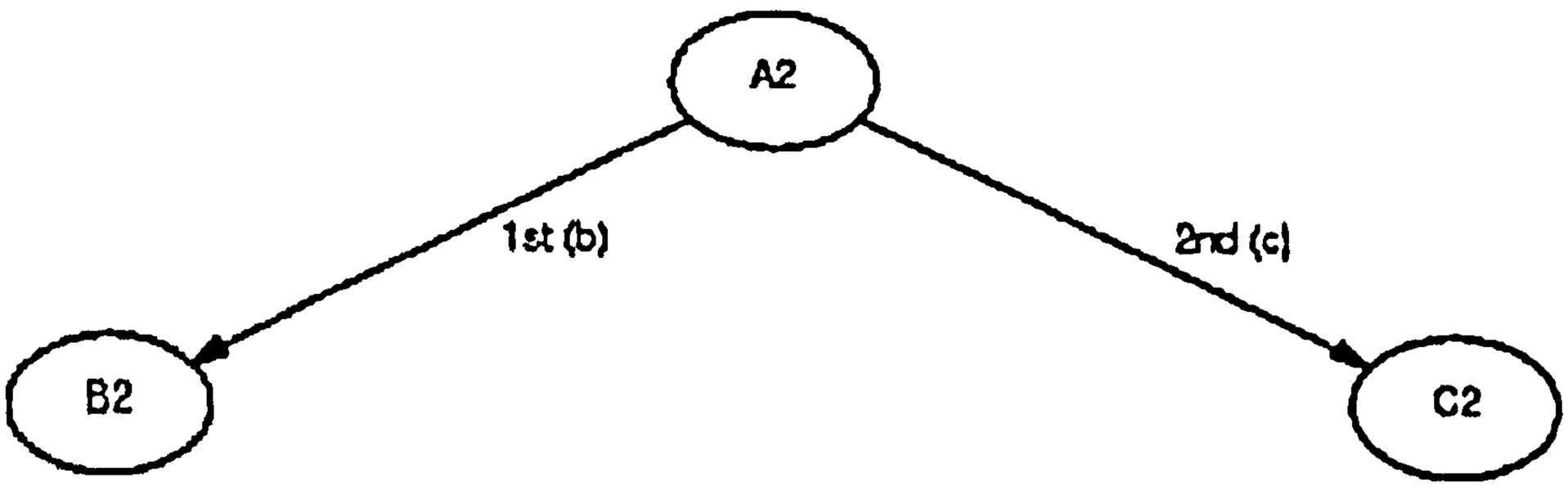

Figure 7.4: Reference Shape for the simple2() method. field.

Program 7.2 A very simple method which invokes one method on each of the class instances referenced by the instance fields of an A2 class instance.

```
class A2{
    private B2 b;
    private C2 c;

    // Constructor
    public A2(){
    b = new B2();
    c = new C2();
    }

    // Dereference fields
    public void simple2(){
    b.traverseB2();
    c.traverseC2();
    }
}
```

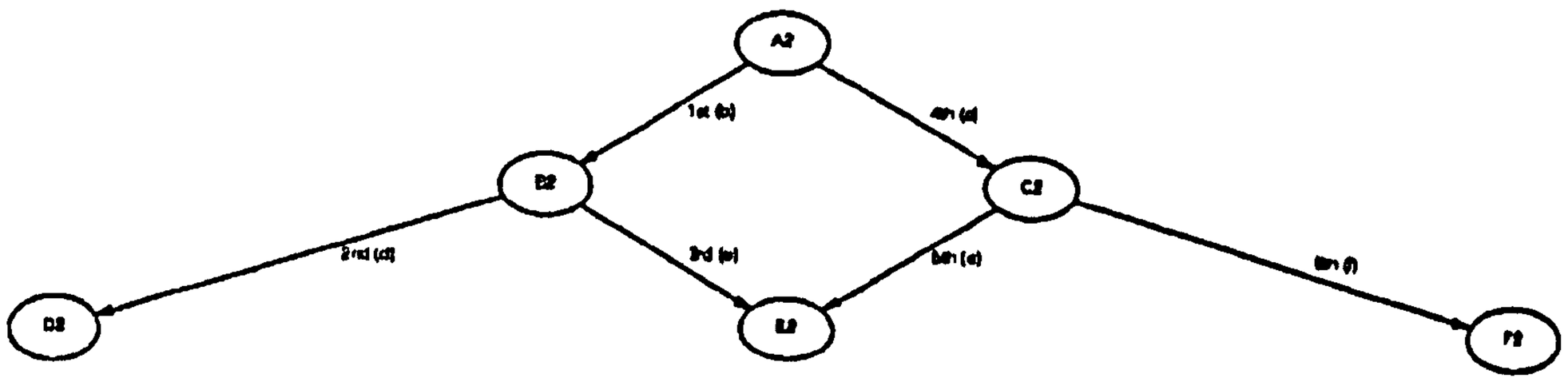

Figure 7.5: Inter-procedural reference shape for the simple2() method.

Program 7.3 .

```
class B2{
    private D2 d;
    private E2 e;

    // Constructor
    public B2(){
    d = new D2();
    e = new E2();
    }

    // Dereference fields
    public void traverseB2(){
    d.doSomethingD2();
    e.doSomethingE2();
    }
}

class C2{
    private E2 e;
    private F2 f;

    // Constructor
    public C2(){
    e = new E2();
    f = new F2();
    }

    // Dereference fields
    public void traverseC2(){
    e.doSomethingE2();
    f.doSomethingF2();
    }
}
```

Method Parameters and Return Values

The examples noted above demonstrate that methods drive the traversal of object graphs. However, in these examples, the traversals were limited to those which took place between a method's receiver object and its fields.

The concept of a reference shape is specific to a particular method. The methods drive the execution of the application, but so far, the analysis presented has only been concerned with methods which traverse the fields of the object which acts as the receiver of the method. In order to more accurately reflect the sequence of object accesses produced when the method executes, this analysis is extended to consider the other ways in which a method might access other objects. The use of parameters in a method introduces another possibility for object traversal, as follows.

References passed as parameters may be accessed in the same way as the receiver object's fields, and hence in a sense may simply be considered as further object fields in the context of a given method. This provides the means to traverse from a receiver object to one of the objects referenced in the actual parameters of a method. Consider program 7.4. Here, the `simple4()` method defined by the `A4` class invokes the `simple4( C4 param )` method on the `b` field, passing the `c` field as a parameter. This method then invokes the `simple4()` method on `param`, the parameter passed in, thereby accessing the fields of an instance of `C4`. The resulting traversal sequence formed by the reference shape for a class instance `a` of `A4` can then be seen as

```
a, a.b, a.b.<simple4(parameter #1)>
```

This would resolve to `a, a.b, a.b.c`.

Methods which return objects provide yet another means for traversal between objects. When a method call expression results in an object being returned to the currently executing method, traversals may occur between the receiver and the returned object. Consider program 7.5. Here, the

Program 7.4 Traversals through method parameters.

```
class A4{
  private B4 b;
  private C4 c;
  .
  .
  .

  public void simple4(){
    b.simple4( c );
  }
}

class B4{
  .
  .
  .

  public void simple4( C4 param ){
    param.simple4();
  }
}

class C4{
  private int i1;
  .
  .
  .

  public simple4(){
    i1 ++;
  }
}
```

sequence of traversals for a class instance a of A5 would be

```
a, a.b,a.<a.b.simple5()> return result
```

This would resolve to `a, a.b, a.b.c`.

Program 7.5 Traversals through method results.

```
class A5{
  private B5 b;
  .
  .
  .

  public void simple5(){
    C5 c = b.simple5();
    c.simple5();
  }
}

class B5{
  private C5 c;
  .
  .
  .

  public C5 simple5(){
    return c;
  }
}
```

In this way, it can be seen that analysis of the method code leads to a more fully defined view of the possibilities for traversal within the object graph than is afforded by schema analysis alone. Although neither traversals through method parameters nor method results use explicitly defined relationships between class instances via the fields of the receiver, it is useful to model these as such in order to more fully capture the behaviour of run-time execution in terms of the objects which will be accessed.

Branching Behaviour

The reference shape of a method is complicated by the conditional and iterative execution of branches within methods.

The basic blocks of a method are punctuated by conditional branches. These constructs introduce a degree of ambiguity to the reference shape of the method, since the referencing behaviour may be predicated upon the evaluation of some conditional expression.

Program 7.6 outlines such a method, `simple3()`, where the reference shape is predicated upon the value of the instance field `decider`. Analysis of the code and schema show that either `b` or `c` will be referenced, but until the data needed to evaluate the conditional expression becomes available, it is impossible to disambiguate the reference shape.

Since orthogonally persistent object systems afford access to the objects populating a store, conditional expressions can sometimes be evaluated statically by examining the state of individual classes and class instances. By applying the reference shape to the object graph in this way, we can specialise the reference shapes obtained through schema and method analysis to dis-ambiguate the path through the object graph.

This process is illustrated in figure 7.6. Here, the reference shape implied by the `simple3()` method (see program 7.6) is applied to two objects of class `A3`, each of which has a different state. One of the `A3` objects has `-1` in its `decider` field, the other has `1`. The method code for `simple3()` shows that the reference shape is ambiguous, in that it may reference either `b` or `c`, however since the expression corresponds directly to a field from an object present in the store, inspection of that object enables the specialisation of the reference shape.

The conditional expression appearing in `simple3()` is very simple, however it is also conceptually possible to have an arbitrarily complex expression which determines the outcome of the branch, and therefore the resulting reference shape. For example, this may be a method call expres-

Program 7.6 The reference shape of the simple3() method is dependent upon the value of the decider field in the class instances of A3.

```
import java.util.Random;

class A3{
    private B3 b;
    private C3 c;
    private int decider;

    // Constructor
    public A3(){
    Random rand = new Random( System.currentTimeMillis());
    b = new B3();
    c = new C3();
    decider = rand.nextInt();
    }

    // Dereference fields
    public void simple3(){
    if( decider > 0 ){
        b.traverseB3();
    } else {
        c.traverseC3();
    }
    }
}
```

Figure 7.6: Application of the reference shape of simple3() to two A3 class instances.

sion yielding a boolean value. In order for the reference shapes of methods involving conditional expressions to be disambiguated, it is necessary to evaluate those conditionals where possible.

7.3 Overview of Sympa

Sympa aims to combine the high prediction accuracy benefits of static code-based analysis with the long lookaheads of static data-based analysis. In demonstrating the orthogonality of fetching and prediction mechanisms, Sympa uses a similar fetching mechanism to that seen in [Kna97b].This is accomplished in three stages, which are carried out when the system is quiescent.

1. Analysis of the class schema and method code of the application to discover a series reference shapes over the classes of the schema.

2. The traversal of the object graph using the established pathways while noting when those pathways cross between different pages. Object relationship data is used to predict which pages will be accessed as the application runs.

3. For each pathway through the object graph which crosses page boundaries, an entry is stored which relates the identifier of the starting object, and the method the reference shape is associated with in a table. This table provides fast lookups at run time of the page answer for methods in relation to particular objects.

The first stage involves the construction of a call multigraph to establish the caller/callee relationships between the methods of the application. Following this, each method represented in the call multigraph is visited to establish the order in which fields, return values and parameters are accessed. The call multigraph is then traversed to produce a larger inter-procedural reference shape which extends across many methods, and classes.

The second stage involves browsing the objects of the store using the reflective interface to find the class of each object in the store. Since the class information defines the set of methods which can be invoked, this information is used to index a registry of inter-procedural reference shapes keyed by method. The appropriate reference shape is then used to navigate through the object graph. As the navigation proceeds, any relationship which crosses two pages is noted and the method associated with the inter-procedural reference shape is recorded in a table with a reference to the first object accessed by the inter-procedural reference shape.

The third stage involves the creation of a two-level hash-table which, at run time, takes an object identifier and a method identifier, and returns an associated list of pages to fetch.

This section discusses the requirements of Sympa's prediction environment before going on to describe the components and processes involved in Sympa's operation.

7.3.1 Sympa's Prediction Environment

Sympa may be ported to another environment which provides the following key features.

- Access to the class schema and method code of an application must be possible in a manner which disregards the declared visibility of class members. That is, public, protected and private fields and method code must be accessible.

- There must be a reflection-based interface to the persistent object store. This interface must support the browsing of the persistent object graph in the store from a root object in such a way that the nature of objects can be determined (eg, classes, interfaces, class instances, arrays). This reflection interface must also disregard the declared visibility of member fields. In this way the relationships between objects specified by private or protected can be traversed.

- There must be a mechanism for communicating with the store layer to tell when navigation

from one object to another in the persistent store would result in a reference to a different page in the store.

7.3.2 The Call Multigraph

Any optimisation or analysis technique which spans more than one method invocation requires an underlying representation of the program structure. The call multigraph is a static structure which describes the dynamic invocation relationships between methods in an application program. A node in the call multigraph represents a method. An edge $a \twoheadrightarrow b$ exists if method a can invoke method b. Such an edge is added to the call multigraph for each call site in a invoking b. Since the call multigraph summarises the relationships between the methods of an application program, it serves as the framework for inter-procedural analysis of the sort performed by Sympa.

Sympa uses a binding call multigraph [CK88]. This specialisation of the call multigraph structure represents richer information by maintaining mapping information on the mapping of actual to formal parameters between each pair of nodes in the call multigraph.

7.3.3 Local Reference Shapes

Each node in the call multigraph maintains a *local reference shape* which captures the order in which fields of the receiver object are accessed. References to non-scalar fields result in a traversal of an object relationship. The order of these traversals is noted in the local reference shape.

In constructing the local reference shape for a method, the method code is analysed to identify statements which access method parameters, instance fields, static fields, and values returned from methods.

In terms of branches in the method code and the related reference shapes possible, a distinction is made between those conditional expressions which:

1. may be resolved in the context of values held in the field of a class instance in the store;

2. may (possibly) be resolved in the context of a subset of the method's callsites;

3. cannot be resolved at all.

In the first case, the prediction of traversal beyond the point of the conditional branch can only take place when applied to the individual objects of the store. This approach assumes that the values of the field will not mutate significantly over time, an assumption common to other static data-based predictions [Kna98, Kna97b].

In the second case, prediction beyond the branch may or may not be possible, depending upon whether the parameters of a parent method in the call multigraph are available.

In the third case, no reliable predictions can be made past the branch. This effectively limits the size of the reference shape.

The view of the method is essentially one which takes account of static behaviour of the method. Consequently, the reference shapes fail to account for looping behaviour. Accordingly, conditional branches which are found to form loops are treated as the third kind of branch above which effectively limits prediction lookahead.

Accesses to method return results, method parameters, and fields are recorded in the local reference shape relative to the current method.

7.3.4 Inter-procedural Reference Shapes

Once all local reference shapes have been built, the call multigraph is traversed in order to construct larger inter-procedural reference shapes from the local reference shapes. This is accomplished by starting from the root node in the call multigraph and visiting the local reference shape of the node and all its descendent nodes. For each local reference shape visited, it is "transposed" and written

to the inter-procedural reference shape such that the elements of the local reference shape are in the form of absolute references to fields.

7.3.5 Applying Inter-procedural Reference Shapes

Once the inter-procedural reference shapes have been constructed, they may be applied to the object graph in the persistent store. The reflection-based browsing interface reads the references and uses them to navigate from the root object of the persistent store. When the navigation of the object graph crosses a page boundary, the page is stored as part of the page answer associated with the methods execution with respect to a particular store object.

7.4 Relation of Sympa to Other Work

Obtaining prediction information from the class schema has been attempted before by Chang [CK89], who used the inheritance hierarchy of the class schema to cluster objects onto pages for those traversals which followed the hierarchy. In this respect, Sympa is similar, since it also relies upon relationships between classes to perform prediction.

Knafla [Kna99] used analysis of object relationships which were present in the form of object fields to perform prediction of page accesses. This approach is similar to Sympa's except that Sympa's model of relationship traversal includes scope for greater prediction coverage by treating objects passed as parameters and returned from method calls as additional object fields which may be traversed.

The concept of a page answer in response to an encapsulated query on a specific object in OODB was proposed by Gerlhof [GK94b]. This approach is similar to that taken by Sympa, since by associating method invocations and objects with page answers, the same assumptions are being made about freedom from side-effects.

Work done in interprocedural optimisation [Hal91] in which an in-lined version of a procedure can be created to subsume the functionality of the original procedure's many descendent procedures. In the field of compiler optimisation, this is employed to eliminate the expense of procedure calls. In Sympa, a similar process is exploited to generate large prediction lookaheads.

7.5 Conclusions

This chapter presented the concept and overview of the Sympa prediction mechanism. This mechanism has not been proposed as the optimal prediction mechanism. Instead, it has been proposed as a hybrid mechanism which exploits the classification (see chapter 4 to obtain the benefits of accuracy of static code-based prediction and prediction lookahead of static data-based mechanisms.

Analysis of OODB benchmarks designed to produce application workloads typical to CAD/CAM applications make use of complex, heterogeneous class schemas. Sympa attempts to use this complexity to generate accurate predictions. Consequently, it will perform well in applications which are based around complex heterogenous class structures and poorly around a simple homogeneous classes like linked lists.

Chapter 8

Evaluation of Sympa

In chapter 7, Sympa was proposed as a hybrid prediction mechanism which exploits the benefits of both data-based and code-based static, codified knowledge. Chapter 5 discussed the need for evaluation in a manner which captures the qualities of a prediction mechanism independently of the many operational parameters which affect performance.

Chapter 5 concluded that although prediction mechanisms from different areas of the classification (presented in chapter 4) are affected by operational parameters in different ways, they all have the same goals of correctly predicting the data requirements of an application in sufficient time to allow predicted data to be fetched.

This chapter uses the Sympa prediction mechanism as the target of evaluation in order to demonstrate the approach to evaluation proposed in chapter 5. The approach is to develop micro-benchmarks which address the fundamental requirements for effective prediction: prediction accuracy, prediction lookahead, and prediction coverage (chapter 5).

8.1 Analysis of Sympa

In developing micro-benchmarks for Sympa, the following questions must be answered with respect to prediction accuracy, prediction lookahead, and prediction coverage.

- What prediction environment does this mechanism require, and how does the mechanism exploit it?
- What are the metrics used in this mechanism?
- What are the optimal and pathological cases of operation?

By answering these questions, it is possible to design micro-benchmarks which expose the strengths and weaknesses of the prediction mechanism. Instead of a potential adopter reading the performance of a prediction mechanism in the context of a specific set of operational parameters, they must examine the performance of Sympa with the micro-benchmarks and ask themselves how closely their target application matches with the characteristics of the micro-benchmarks.

Requires a prediction environment with the following characteristics.

- Access to the class schema and method code of an application must be possible in a manner which disregards the declared visibility of class members. ie public, protected and private fields and method code must be accessible.
- There must be a reflection-based interface to the persistent object store. This interface must support the browsing of the persistent object graph in the store from a root object in such a way that the nature of objects can be determined (eg classes, interfaces, class instances, arrays). This reflection interface must also disregard the declared visibility of member fields. In this way the relationships between objects specified by private or protected can be traversed.

- There must be a mechanism for communicating with the store layer to tell when navigation from one object to another in the persistent store would result in a reference to a different page in the store.

The way in which Sympa exploits the prediction environment is discussed in chapter 7. Briefly, the mechanism works at two levels. Firstly, the class schema and method code are analysed to produce inter-procedural reference shapes over the fields, parameters and return values relative to a root method. Secondly, this inter-procedural reference shape is used to navigate the object graph in the persistent store, and thus predict object and page accesses. Where the inter-procedural reference shape is ambiguous due to the presence of a conditional expression, and the conditional expression relates to a field of a persistent object, then the value from the field is used to disambiguate the reference shape on the basis of individual objects.

The metrics for the mechanism are simple. Percentages for prediction accuracy and prediction coverage should be sufficient. Given that the unit of prediction at the level of the inter-procedural reference shape is references relative to a base object and method, the metric for lookahead should be the number of references to non-scalar fields which can be predicted.

8.2 Prediction Accuracy

This section addresses prediction accuracy. Specifically, the focus is upon the degree of overlap between the predictions offered by the prediction mechanisms and the data requirements of the application.

The method analysis stage of Sympa which results in production of the inter-procedural reference shape uses static, codified knowledge based on the code of the application program. In common with other code-based prediction mechanisms, it has the advantage of 100% prediction accuracy over the references predicted.

The reference shape application stage of Sympa applies the inter-procedural reference shapes to the object graph in the persistent store. This stage uses the values of fields comprising the state of persistent objects in order to partially evaluate conditional expressions (where possible) and thereby disambiguate the reference shapes produced by branches in the method code.

Applying the inter-procedural reference shapes in this way produces predictions in terms of a series of objects (and therefore pages) which would be accessed from the persistent store. However, the accuracy of the predictions at this level are predicated upon the assumption that the majority of field values will not change. Crucially, the perfect prediction of the inter-procedural reference shapes is compromised here.

The micro-benchmark shown in program 8.1 shows the way in which the inter-procedural reference shape of the `traverse()` method of `Part3` can be disambiguated on the basis of field values of objects in the persistent store. At the level of the inter-procedural reference shape, the prediction accuracy is perfect, but the prediction coverage is limited (as discussed in section 8.4). The elements of the reference shape which are unknown as a result of the ambiguity caused by branches are represented here as <unknown>. The predicted references in relation to `Part3` and the method `traverse()` are shown below.

```
this, <unknown>, this.pb, <unknown>, this.pb.b
```

If the inter-procedural reference shape were to be applied to a class instance of `Part3` present in the store which had the `private boolean follow` field set to false and a related `Part4` class instance with its field set to false, the resulting stream of references would be as follows.

```
this, this.pb, this.pb.b
```

With the application of the inter-procedural reference shape to the object graph, the accuracy of this prediction is dependent upon whether or not the field retains its value. The pathological case

for accuracy in an applied reference shape over objects is where the conditional expression depends upon a constantly changing value.

8.3 Prediction Lookahead

For the inter-procedural reference shapes, prediction lookahead can be assessed in terms of the number of references to non-scalar fields which can be detected.

The micro-benchmark shown in program 8.2 exhibits a class schema consisting of a single class and demonstrates the pathological case for prediction lookahead. The inter-procedural analysis of the methods of this class reveal that the greatest lookahead is possible within the reference shape associated with the `getTail()` method. However, even this yields only a single reference lookahead.

```
this, this.tail
```

Clearly, the smaller and simpler that class schema, the smaller the prediction lookaheads possible with Sympa.

The micro-benchmark shown in program 8.3 shows that with more complex class schemas, longer lookaheads are possible. Considering the `traverse()` method of `Part1`, the reference shape has a prediction lookahead 6 references long.

```
this, this.pa, this.pa.a, this.pa.b, this.pb, this.pb.a, this.pb.b
```

The optimal case for prediction lookahead at the level of the inter-procedural reference shape is a lookahead that is limited only by the size of the class schema traversed by the method at the root of the shape and its descendents.

Program 8.1 Conditional branches based on field values.

```
class Part3{
  private Part4 pa;
  private Part4 pb
  private boolean follow;

  Part3( boolean follow ){
    this.follow = follow;
    pa = new Part4();
    pb = new Part4();
  }

  public void traverse(){
    if( follow ){
      pa.traverse();
    }
    pb.traverse();
  }
}

Class Part4{
  private Object a;
  private Object b;
  private boolean follow;

  Part4( boolean follow ){
    this.follow = follow;
    a = new Object();
    b = new Object();
  }

  public void traverse(){
    if( follow ){
      a.hashCode();
    }
    b.hashCode();
  }
}
```

Program 8.2 Minimal class schema.

```
class List{
  private List tail;
  private int data;

  public void setData( int data ){
    this.data = data;
  }

  public int getData(){
    return data;
  }

  public void setTail( List tail ){
    this.tail = tail;
  }

  public List getTail(){
    return tail;
  }
}
```

Program 8.3 Complex class schema.

```
class Part1{
  private Part2 pa;
  private Part2 pb

  Part1(){
    pa = new Part2();
    pb = new Part2();
  }

  public void traverse(){
    pa.traverse();
    pb.traverse();
  }
}

Class Part2{
  private Object a;
  private Object b;

  Part2(){
    a = new Object();
    b = new Object();
  }

  public void traverse(){
    a.hashCode();
    b.hashCode();
  }
}
```

8.4 Prediction Coverage

Prediction coverage is concerned with the proportion of the application's total references for which predictions can be made.

With reference to the micro-benchmark presented in program 8.4, the percentage of the application's references which may be predicted is limited by the status of the conditional expression. If, through inter-procedural analysis, the value of the boolean parameter `follow` can be related to either a constant, or instance field, then full coverage of the application's references can be made. In the case where the value of `follow` cannot be determined, the conditional expression acts as a barrier to further coverage of the application's references following the branch.

At the level of the inter-procedural reference shape, the optimal and pathological cases of prediction coverage mirror those of prediction lookahead.

8.5 Conclusions

This chapter represents a rather hasty and minimal evaluation of the Sympa prediction mechanism. It provided a demonstration of the approach to evaluation which was outlined in chapter 5. To be of more use in exposing the qualities of Sympa, it needs to be expanded.

In order to completely justify the evaluation mechanism, these benchmarks would be compared against a real system. This is left as further work at this stage.

Program 8.4 Conditional branches based on unknown values.

```
class Part5{
  private Part6 pa;
  private Part6 pb

  Part5(){
    pa = new Part6();
    pb = new Part6();
  }

  public void traverse( boolean follow ){
    if( follow ){
      pa.traverse();
    }
    pb.traverse();
  }
}

Class Part6{
  private Object a;
  private Object b;

  Part6( ){
    a = new Object();
    b = new Object();
  }

  public void traverse( boolean follow){
    if( follow ){
      a.hashCode();
    }
    b.hashCode();
  }
}
```

Chapter 9

Conclusions

This thesis presented the hardware trends which make prefetching an important area of research across many application areas, ranging from in-core scientific applications to pre-emptive push web servers. The approach has been to present a separate analysis of the mechanisms which constitute prefetching schemes. One of the main contributions which resulted from this approach was the creation of a classification of prediction and fetching mechanisms and subsequently, a taxonomy into which extant predictors and fetchers were placed.

Through consideration of prediction mechanisms across the taxonomy, the factors which influenced their effectiveness in terms of prediction accuracy and lookahead was drawn up. This was used to present the other main contribution of this work: an approach to evaluation which addresses the quality of the prediction mechanism itself in a manner which exposes the relative advantages and disadvantages of a mechanism.

Another contribution of the thesis was the design of the Sympa prediction mechanism for object oriented persistent systems. This hybrid prediction mechanism was inspired by the breadth of the prediction mechanism taxonomy. Sympa aims to offer better prediction accuracy than prediction mechanisms applied to a similar area [GK94b, PZ91, CKV93, Kna99] by incorporating elements

from different areas of the taxonomy to achieve the benefits of both. In keeping with the general findings of the thesis, Sympa works best in object oriented applications displaying specific characteristics, notably large, complex class schemas.

In order to demonstrate the approach to evaluation proposed by the thesis, an evaluation of Sympa is presented.

This chapter gives more details on the conclusions of the research.

9.1 The Importance of Prefetching

This thesis showed that the growth trends in both CPU performance relative to memory performance and in memory performance relative to magnetic disk performance are diverging. In particular, while magnetic disk bandwidth is increasing at a modest rate, the improvement in magnetic disk latency is beginning to plateau as a result of the engineering constraints of magnetic disks.

While latency reduction optimisations such as caching and clustering attempt to reduce the number of high-latency read operations, the initial cost of those operations still has an impact on the total execution time of an application. As a latency tolerance optimisation, prefetching attempts to avoid even this cost by overlapping fetching operations for soon to be needed data items with the ongoing execution of the application program. As such, prefetching appears to be the most promising approach to latency optimisation in cases where device bandwidth is available in excess, and the data requirements can be predicted early enough to make the necessary data resident ahead of its use by the application program. Accordingly, the prediction element of prefetching is a key concern in the development of prefetching schemes which result in a reduction in execution times.

9.2 Fetching Predicted Data

This thesis presented prefetching as consisting of two separate mechanisms. This reflects the efforts of others [GK94a, MDK96] in porting prediction mechanisms to other fetching mechanisms. The mechanisms are:

1. prediction mechanisms to prediction an application's future data requirements

2. fetching mechanisms to make predicted data resident in a lower latency level of the memory hierarchy.

The fetching mechanisms were broadly classified into those which use explicit fetching and those which use indirect fetching. The primary difference between the two being that although, as in explicit prefetching, data is made resident in parallel with the application's execution, indirect fetching mechanisms include checks to ensure that prefetching data is likely to improve performance.

There are many factors which can lead to explicit fetching mechanisms degrading performance of applications, since the prefetch will be performed regardless of:

- the presence of requested items in the cache

- the demand for memory, disk, or network bandwidth

- the effect of multi-user or multi-threaded loads on the service times for prefetches.

While it is clear that the overheads of indirect fetching mechanisms are higher than those of explicit mechanisms, it appears that the additional overhead costs are more than covered by the resulting improvements in execution time, if not by reduced cache misses.

Ultimately, the performance of a fetching mechanisms is dependent upon both the support from the platform and the nature of the predictions: for example whether the predictions are of cache misses or of references.

9.3 The Prediction of Data Requirements

This thesis proposed a classification of prediction mechanisms and placed extant prediction mechanism into a taxonomy build upon the classification. This taxonomy included prediction mechanisms which spanned multiple application areas and which addressed different latency barriers. Through the classification and taxonomy, the fundamental requirements, similarities and goals of prediction mechanisms were drawn together.

The key to prediction is to make assumptions on the operation of an application program. These assumptions can take the form of tacit knowledge obtained through observation of the application, or through codified knowledge obtained through analysis of the application's semantics.

The stronger the statement one can make about the operation of an application program, the greater the potential for long, accurate predictions.

This would seem to advocate a style of software engineering in which as much information as possible was available to the prediction environment. If everything that could affect the behaviour of the application was present in the store, then the behaviour can be predicted very accurately [GK94b]. The user interaction provides external influences which cannot be predicted. The logical conclusion is therefore in accord with the approach of orthogonally persistent programming languages [AM95] in which software is developed, run and maintained within a single software environment.

9.4 Meaningful Evaluation

This thesis reflected the effort undertaken in the search for a universal evaluation method and set of metrics for prediction mechanisms. In the course of this, the use of analysis techniques including simulation and mathematical models as well as the traditional approach of direct measurement were

critically examined. The analysis of the different factors affecting the performance of a prediction mechanism (highlighted during the formation of the taxonomy and classification) led to the conclusion that no such universal evaluation method could exist. Instead, the problem of evaluation was posed via an alternate route by acknowledging the similar goals of each prediction mechanism. Bespoke micro-benchmarks were proposed to highlight the best and worst cases for prediction accuracy and lookahead and examples were given. The approach advocated the use of separate micro-benchmarks for separate classes of prediction mechanism. These classes of prediction mechanism corresponded to the classification appearing in chapter 4, since each mechanism within a classification had broadly similar requirements of their environments and were affected in similar ways. The exception to this is the set of strategy-based mechanisms. The only common feature among these mechanisms is that the prediction was done by the designer of the prefetching scheme on the basis of some property of the prediction environment. For example, noticing that file systems tend to access files sequentially from first to last block. The lack of common features among strategy-based prediction mechanisms necessitates the production of micro-benchmarks for each individual mechanism within this part of the classification.

It seems that as far as evaluation is concerned, before performance evaluation between prediction mechanisms can take place, we need to find some way of comparing like with like. Since prediction mechanisms approach prediction in different ways this is very difficult. The goal of all prefetching schemes is the same: to reduce execution time. The goal of all prediction mechanisms is the same: to predict data requirements accurately and with the longest possible lookahead. However, because they are implemented in such different ways, there can be no single set of either metrics or tests which could be run against all prediction mechanisms to find the optimal one. For example, the size of the data footprint of an application used to test the efficacy of a training based prediction mechanism is appropriate. It is not appropriate to use the same test for a code-based mechanism,

since varying the size of the workload will not affect the performance. Instead, each mechanism needs to show the cases where it performs best and worst. The benchmarks are used to address the different aspects of prediction: accuracy and lookahead. There are subdivisions within this to cope with coverage. Within each subdivision there are benchmarks to address the performance of prediction mechanisms in that part of the classification. So the classification system has helped in creating the evaluation.

Sympa was introduced not as a universally optimal prediction mechanism, since such an assertion would be contrary to the findings of the work done in evaluation which proposes that there is no such thing as the optimal prediction mechanism. Instead, Sympa was introduced as a mechanism which took elements from across the breadth of the taxonomy to provide the prediction accuracy benefits of static code-based analysis and the lookaheads of static data-based analysis.

9.5 Further Work

Having produced a classification of prediction mechanisms along the dimensions presented in chapter 4, one possible area of future research is to identify areas within the classification in which no prediction mechanisms exist.

The approach of using micro-benchmarks targeted at either individual prediction mechanisms or groups of similar mechanisms was proposed in chapter 5 and demonstrated in chapter 6. A useful contribution to prefetching would be the development of more benchmarks targeted at other prediction mechanisms and groups of similar prediction mechanisms. From the perspective of software engineers, this would make the process of choosing an appropriate prediction mechanism simpler.

This thesis supported the concept of orthogonality between prediction and fetching mechanisms. Yet another avenue for future research would be the investigation of alternative fetching mechanisms for the Sympa prediction mechanism. In particular, it would be interesting to apply the inter-

procedural reference shapes of Sympa to create specialised "in-lined" versions of methods with explicit prefetch statements planted automatically in the code. This would seem to be a promising way of addressing the latency of object transfers from page-based in-memory representations to heap representations ready for computation.

Bibliography

[ADJ+96] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4), December 1996.

[AK97] Jung-Ho Ahn and Hyoung-Joo Kim. SEOF: An Adaptable Object Prefetch Policy for Object-Oriented Database Systems. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 4–13, Birmingham, UK, April 1997.

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

[Atk78] Malcolm P. Atkinson. Programming Languages and Databases. In *Fourth International Conference on Very Large Data Bases*, pages 408–419, West Berlin, Germany, September 1978. IEEE-CS.

[Bes95] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of the 1995 conference on International conference on information and knowledge management*, pages 403–410, Baltimore, MD USA, November 1995. ACM.

[BKW94] Kavita Bala, Frans M. Kaashoek, and William E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *Proceedings of the First Symposium on*

Operating System Design and Implementation, pages 243–253, Monterey, CA, November 1994.

[BS96] Frédérique Bullat and Michel Schneider. Dynamic Clustering in Object Databases Exploiting Effective Use of Relationships Between Objects. *Lecture Notes in Computer Science*, 1098:344–365, 1996.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *Proceedings of the 1993 ACM SIGMOD Conference on the Management of Data*, volume 22, pages 12–21, Washington DC, USA, June 1993.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 188–197, 1995.

[CFKL96] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling. *ACM Transaction on Computer Systems*, November 1996.

[CH91] Jia Bing R. Cheng and A. R. Hurson. Effective Clustering of Complex Objects in Object-Oriented Databases. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, volume 20, pages 22–31, Denver, Colorado, May 1991.

[Cha89] Ellis E-Li Chang. *Effective Clustering and Buffering in an Object-Oriented OODBMS*. PhD thesis, University of California, Berekeley, 1989.

[CK88] K. D. Cooper and K. Kennedy. Interprocedural Side-Effect Analysis in Linear Time. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, USA, 1988. ACM, ACM Press.

[CK89] Ellis E. Chang and Randy H. Katz. Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS. In *Proceedings of the 1989 SIGMOD International Conference on the Management of Data*, pages 348–357, Portland, Oregon, June 1989.

[CKV93] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In *Proceedings of the 1993 ACM SIGMOD*, volume 22, pages 257–266, Washington, DC, June 1993. ACM, ACM Press.

[Den80] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, se-6(1):64–84, January 1980.

[DPS98] Jérôme Darmont, Bertrand Petit, and Michel Schneider. OCB: A Generic Benchmark to Evaluate the Performances of Object-Oriented Database Systems. In *6th International Conference on Extending Database Technology (EDBT '98)*, number 1377 in LNCS, pages 326–340, Valencia, Spain, March 1998. Springer-Verlag.

[DS99] Jérôme Darmont and Michel Schneider. VOODB: A Generic Discrete-Event Random Simulation Model To Evaluate the Performances of OODBs. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 254–265, Edinburgh, Scotland, September 1999. Morgan Kaufmann.

[FCL93] Michael Franklin, Michael Carey, and Miron Livny. Local Disk Caching for Client-Server Database Systems. In *Proc. 20th Intl. Conference on Very Large Data Bases (VLDB)*, Dublin, Ireland, August 1993.

[GA94] James Griffioen and Randy Appleton. Reducing File System Latency using a Predictive Approach. In *Proceedings of the 1994 Summer USENIX Conference*, pages 8–12, June 1994. Also released as TRCS247-94, University of Kentucky.

[GAN93] Knut Stenner Grimsrud, James K. Archibald, and Brent E. Nelson. Multiple Prefetch Adaptive Disk Caching. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):88–103, February 1993.

[GK94a] Carsten A. Gerlhof and Alfons Kemper. A Multi-Threaded Architecture for Prefetching in Object Bases. In *Proceedings of the 4th International Conference on Extending Database Technology (EDBT)*, volume 779 of *Lecture Notes in Computer Science (LNCS)*, pages 351–364, Cambridge, England, March 1994. Springer-Verlag.

[GK94b] Carsten A. Gerlhof and Alfons Kemper. Prefetch Support Relations in Object Bases. In *Proc. of the 6th Intl. Workshop on Persistent Object Systems (POS)*, Workshops in Computing Series (WICS), pages 115–126, Tarascon, Provence, September 1994. Springer-Verlag.

[GKKM92] Carsten A. Gerlhof, Alfons Kemper, Christoph Kilger, and Guido Moerkotte. Partition-Based Clustering in Object Bases: From Theory to Practice. In *Proc. of the 4th Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, volume 730 of *Lecture Notes in Computer Science (LNCS)*, pages 301–316, Chicago, Illinois, October 1992. Springer-Verlag.

[GLC+92] Y. Gourhant, S. Louboutin, V. Cahill, A. Condon, G. Starvoic, and B. Tangney. Dynamic Clustering in an Object-Oriented Distributed System. In *Proceedings of the OLDA-II (Objects in Large Distributed Applications)*, Ottawa, Canada, October 1992.

[Hal91] Mary Wolcott Hall. *Managing Interprocedural Optimisation*. PhD thesis, Rice University, Houston, Texas, USA, 1991.

[HMMS95] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing Loads: Enabling Software to Observe and React to Memory Behavior. Technical Report CSL-TR-95-673, Stanford University, Stanford University, July 1995. Also numbered STAN-CS-95-675.

[HMMS98] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Application. *ACM Transactions on Computer Systems*, 16(2):170–205, May 1998.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 2nd edition, 1996.

[Jos70] M. Joseph. An Analysis of Paging and Program Behaviour. *Computer Journal*, 13(1):48–54, February 1970.

[JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2000.

[KE90] David F. Kotz and Carla Schlatter Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, 1990.

[KGM91] Tom Keller, Goetz Graefe, and David Maier. Efficient Assembly of Complex Objects. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 148–157, Denver, Colorado, May 1991.

[KKP94] Ken Kennedy, Charles Koelbel, and Mike Paleczny. Scalable I/O for Out-of-Core Structures. Technical Report CRPC-TR93357-S, Center for Research and Parallel Computation, Rice University, Rice University, August 1994. Previous version published in November 1993.

[Kna97a] Nils Knafla. A Prefetching Technique for Object-Oriented Databases. In *Advances in Databases*, 15th British National Conference on Databases, pages 154–168, London, United Kingdom, July 1997. Springer Verlag.

[Kna97b] Nils Knafla. Predicting Future Page Access by Analysing Object Relationships. Technical Report ECS-CSG-35-97, University of Edinburgh, December 1997.

[Kna97c] Nils Knafla. Speed Up Your Client with Adaptable Multithreaded Prefetching. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 102–111, Portland, Oregon, U.S.A, August 1997. IEEE Computer Society.

[Kna98] Nils Knafla. Private Communication with Nils Knafla, July 1998.

[Kna99] Nils Knafla. *Prefetching Techniques for Client/Server, Object Oriented Database Systems*. PhD thesis, University of Edinburgh, 1999.

[Lam88] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[LD92] Daniel F. Lieuwen and David J. DeWitt. A Transformation-Based Approach to Optimizing Loops in Database Programming Languages. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 91–100, San Diego, California, June 1992.

[Li92] Qing Li. A Conceptual Model for Dynamic Clustering in Object Databases. In *Proceedings of the 18th VLDB Conference*, pages 457–468, Vancouver, Canada, August 1992.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[MDK96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 3–17, October 1996.

[MJLF84] M. K. McKusic, W. J. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for Unix. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK94] William J. McIver and Roger King. Self-adaptive, on-line reclustering of complex object data. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 407–418, Minneapolis, Minnesota, May 1994.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference*

on Architectural Support for Programming Languages and Operating Systems, pages 62–73, October 1992.

[Mow94] Todd C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, March 1994.

[MT93] S.P. Meyn and R.L. Tweedie. *Markov Chains and Stochastic Stability*. Springer-Verlag, 1993.

[PGG+95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium of Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[PH94] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface*. Morgan Kaufmann, 1994.

[PZ91] Mark Palmer and Stanley B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 255–264, Barcelona, Catalonia, Spain, September 1991.

[RPASA97] Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, and Sarita V. Adve. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 144–156, Denver, Colorado, June 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[SC97] V. Srinivasan and D. T. Chang. Object Persistence in Object-Oriented Applications. *IBM Systems Journal*, 36(1), 1997.

[Sch90] Thomas J. Schriber. *An Introduction to Simulation Using GPSS/H*. Wiley, 1990.

[sia99] International Technology Roadmap for Semicondictors. Technical report, Semiconductor Industry Association, 1999.

[Smi78] Alan Jay Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[Sun97] Sun Microsystems. *SPARC Assembly Language Reference Manual*, 1997.

[TN92] Manolis M. Tsangaris and Jeffrey F. Naughton. On the Performance of Object Clustering Techniques. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 91–100, San Diego, California, June 1992.

[TPG97] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '97)*, Seattle, Washington, June 1997. ACM.

[Tri76] Kishor S. Triviedi. Prepaging and Applications to Array Algorithms. *IEEE Transactions on Computers*, 25(9):915–921, September 1976.

[WKM94] Paul R. Wilson, Sheetal Kakkad, and Shubhendu S. Mukherjee. Anomalies and Adaptation in the Analysis and Development of Prefetching Policies. *Journal of Systems and Software*, 27(2):147–153, November 1994.