Client Cache Management in a Distributed Object Database

by

Mark Stuart Day

B.S.C.S., Washington University (1983)
B.S.E.E., Washington University (1983)
S.M., Massachusetts Institute of Technology (1987)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author . . . .
Department of Electrical Engineering and Computer Science
February 23, 1995

Certified by . . . .
Barbara Liskov
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by . . . .
F.R. Morgenthaler
Chair, Department Committee on Graduate Students

Keywords: swizzling, edge marking, node marking, surrogates, shrinking, garbage collection, page fetching, object fetching, performance.

This work was supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136, in part by the National Science Foundation under Grant CCR-8822158, and in part by a graduate fellowship funded by the McDonnell Douglas Corporation.

Client Cache Management in a

Distributed Object Database

by

Mark Stuart Day

Submitted to the Department of Electrical Engineering and Computer Science
on February 23, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A distributed object database stores objects persistently at servers. Applications run on client machines, fetching objects into a client-side cache of objects. If fetching and cache management are done in terms of objects, rather than fixed-size units such as pages, three problems must be solved:

1. which objects to prefetch,
2. how to translate, or *swizzle*, inter-object references when they are fetched from server to client, and
3. which objects to displace from the cache.

This thesis reports the results of experiments to test various solutions to these problems. The experiments use the runtime system of the Thor distributed object database and benchmarks adapted from the Wisconsin OO7 benchmark suite.

The thesis establishes the following points:

1. For plausible workloads involving some amount of object fetching, the prefetching policy is likely to have more impact on performance than swizzling policy or cache management policy.
2. A simple breadth-first prefetcher can have performance that is at worst roughly comparable to an approximation of a paged system, and is sometimes much better, completing some computations 10 times faster.
3. The client cache is managed by a combination of two simple schemes, one based on garbage collection and one based on *shrinking* some objects to *surrogates*, where a surrogate is a small data structure containing the object's name at the server. Garbage collection and shrinking have complementary strengths. Experiments show that cache management using both techniques performs much better than cache management using only one technique. A system with both garbage collection and shrinking can complete a benchmark traversal using roughly an order of magnitude less cache storage than a system using only garbage collection.
4. For minimizing elapsed time on the benchmarks used, managing the client cache with a true LRU policy achieves close to the performance of an unattainably good minimal-fetching policy. The other policies tested all do worse than LRU, and there is little meaningful difference among them on the benchmarks used.
5. Swizzling at the granularity of individual references (so-called edge marking) has performance that is roughly comparable to the performance of swizzling at the granularity of objects (so-called node marking). Accordingly, a choice between these swizzling mechanisms should be made on the basis of factors other than performance, such as implementation complexity.

Thesis Supervisor: Barbara Liskov
Title: NEC Professor of Software Science and Engineering

Acknowledgements

I am grateful to my advisor, Barbara Liskov. I remain in awe of her technical thinking and writing, and I am grateful that I could spend the last several years learning from her example.

I am similarly indebted to the members of my committee: Frans Kaashoek, Butler Lampson, and Jerry Saltzer. It has been a rich, rewarding, and very educational (if occasionally difficult) experience.

David Gelernter and Suresh Jagannathan have provided inspiration and support for a number of years, always providing a stimulating and challenging perspective for which I am very grateful.

Various advisors, supervisors, and mentors have been important in getting me to this point although they did not contribute directly to this thesis. Noteworthy are Catalin Roman, Dave Gifford, Gerry Sussman, Dick Rubinstein, Steve Ward, and Irene Greif in computer science; and Toshi Katayama in graphic design.

A number of graduate student peers have contributed to this work, often at considerable cost to their own endeavors. Andrew Myers, Umesh Maheshwari, Sanjay Ghemawat, Atul Adya, Quinton Zondervan, Phil Bogle, Miguel Castro, Jim O'Toole, Franklyn Turbak, and Mark Sheldon have all dealt with various drafts and questions. Jim has always been a particularly acute and astute critic. I also owe a debt of gratitude to Sanjay and Andrew for their role in working with me on the initial design and coding of the Thor implementation. Ziggy Blair, Jonathan Rees, Sharon Perl, Daniel Jackson, and Mark Reinhold have all offered good advice at various times.

I am grateful to a number of people for explaining aspects of their related work to me: Laurent Amsaleg, Bob Bretl, Jeff Chase, Brad Chen, Mike Franklin, Tony Hosking, Norm Hutchinson, Eric Jul, Ted Kaehler, Eliot Moss, Scott Nettles, Allen Otis, Mark Palmer, Margo Seltzer, Marc San Soucie, Jim Stamos, and Paul Wilson.

Thanks to Mark Glickman, Dave Farrens, and Dan Walczyk for being an amazing Buckaroo-Banzai-style combination of rock music and PhD-level science. I don't know how many other rock bands would have been as sympathetic and supportive of my thesis misadventures while also being able to produce an amazing sound.

My family and friends have been very supportive, and have done very well at learning not to keep asking "so when will you be done?".

I am infinitely grateful that I am still married to Thu-Hăng Trân. She has displayed superhuman tolerance and cheerfulness under perenially difficult circumstances. Anh vân còn yêu mình, y'know?

Finally, special thanks to my children Jefferson Việt-Anh and Eleanor Tâm-Anh. Their enthusiastic persistence in learning has been inspiring.

Chapter 1

Introduction

Some application programs store and share persistent data that consists of small entities with complex relationships. For example, computer-aided design (CAD) applications typically manipulate large collections of data structures describing mechanical or electrical parts. These collections of data must be made available to other applications that determine costs, order supplies, or check feasibility of designs. An *object database* is useful for storing and sharing such collections of parts and their associated relationships, such as which parts together form another part, which parts come from the same supplier, or which parts are made of the same material.

For our purposes, an *object* is an entity that has two properties: it has an identity distinct from its value, and it can contain references to other objects. Identity and references make it possible to build complex data structures involving sharing and cycles, which can be useful for modelling complex application domains. In contrast to typical file systems or relational databases, an object database can directly represent sharing of objects.

An organization may need to share an object database over a large geographic area. To build such a *distributed object database*, we divide the infrastructure into *servers* and *clients*. Servers are specialized to hold data reliably and to provide it to clients on request, while clients are specialized to run applications and present information to people. A server needs a large amount of disk storage. It may be implemented with multiple disks for reliability [43], or with multiple replicated computers to provide both high availability and high reliability [47]. With the cooperation of clients, servers implement concurrency control mechanisms [46] to ensure that groups of objects are updated consistently. In contrast to a server, a client needs a good display, and processing power and storage adequate to the application(s) being run by any particular user.

Dividing the system into clients and servers has a number of advantages. The clients are typically personal machines, and each client can be sized appropriately for the tasks that each person needs to perform. The servers are shared resources that can be carefully managed and physically isolated to protect the stored data from hazards. Overall, this architecture provides for the sharing of data but does not otherwise require sharing resources for application computing. It is attractive for an organization where shared data is used for a variety of purposes.

Caching is a frequently-used technique in systems for improving performance. This thesis considers the problems of using a particular approach to caching objects at the client in such a client/server object database. There are two main approaches that could be used for fetching the objects and managing the cache. The approach studied in this thesis (called *object caching*) simply fetches objects of various sizes from the server and manages the cache in terms of those objects. While an object is being used by a client, a copy of the object is stored in a cache at the client. As with other caches (such as virtual memory page caches or processor caches), the client object cache exploits locality in the computation: an object used once may well be used again shortly thereafter, so that it is more efficient to retain a copy than to refetch it from the server. The other approach (called *page caching*) clusters objects into fixed-size pages, then fetches and evicts pages rather than objects.

Page caching has been the most common approach for implementing distributed object databases. With page caching, the mechanisms for fetching and cache management are simple because the units handled are all the same size; however, a page caching system must have its objects clustered into pages, and that clustering process is difficult [71]. A clustering of objects into pages is a static prediction of overall application access patterns. Because of the dependence on clustering, a page caching system can have poor performance because an application's access pattern changes rapidly, because an application's access pattern is difficult to predict, or because multiple applications have conflicting access patterns for shared objects. Recent work by McIver and King [50] recognizes these problems and proposes *on-line reclustering* as a solution. This thesis proposes, implements, and measures a more radical approach: abandoning the use of pages as the unit of transfer between server and client.

Object caching has been studied less than page caching, and although object caching requires more complex fetching and cache management than page caching, it has the potential to perform better. Intuitively, one would expect object caching to be better than page caching in terms of cache utilization and prefetching flexibility.

- Cache utilization should be higher because only relevant objects need to be fetched into the cache.

- Prefetching can be more effective because a variable-sized group of objects can be selected at run time, taking advantage of hints from the application and recent computational history.

This flexibility means that two applications that use some shared objects in very different ways can each get better performance from the server than they would if there were only a single clustering of objects into pages.

Ultimately, research studies like this one will lead to an understanding of whether object caching or page caching is a better organization for a distributed object database. However, before such a comparison can be made, it is necessary to understand both how to manage an object cache and what the important parameters are for its performance. Copying objects to a client cache and managing that cache introduce new problems. Others have addressed aspects of storage management [48] and concurrency control [1]. The thesis addresses the following three performance-related problems:

- implementing inter-object references;
- bringing objects into the client cache; and
- evicting objects from the client cache.

The most important results are the following:

1. Varying the technique for bringing objects into the client cache has more impact on performance than varying the technique for evicting objects from the client cache.
2. Fetching a dynamically-computed group of objects works as well as or much better than fetching a statically-computed page of objects. Simple breadth-first prefetching based on the connectivity of objects performs well.
3. An accurate but cheap-to-implement least-recently-used (LRU) policy is close to optimal for a workload with a shifting working set. However, a single-bit approximation of LRU (called CLOCK) performs no better than simpler policies that keep no usage information.
4. Different techniques for implementing in-memory inter-object references have similar overall performance. This result contrasts with some previous reports.

1.1 Problems Addressed

The contributions of this thesis fall into three categories: how objects refer to one another in the cache; which objects are brought over to the cache when there is a cache miss; and how to remove objects from the cache so as to make room for new objects.

1.1.1 How objects refer to one another

Following common terminology, I distinguish between *names* and *addresses*. An address is a “direct” or “primitive” reference, while a name involves some level(s) of interpretation and indirection. This terminology is context-sensitive: an entity that is considered a primitive address in one context may be treated as a name that is interpreted to produce a “more primitive” address in another context.

For the systems considered in this thesis, the primitive addresses are hardware virtual memory addresses. This thesis ignores the actual structure and implementation of such virtual addresses. Persistent objects do not refer to one another using virtual addresses when stored at a server; one reason is that such inter-object references can cross machine boundaries. We do not know whether it will ever be possible (or desirable) to build a single shared address space for many thousands of machines, but no-one presently knows how to do so. So a reference from one object to another is implemented as a name when the objects are stored at a server.

Names can also be used for inter-object references when the objects are cached at the client. However, using names makes it quite expensive to follow any inter-object reference. Whenever an inter-object reference needs to be followed, the stored name must be interpreted, yielding a memory address. An alternative approach, called *swizzling*, replaces a name with the corresponding memory address, effectively memoizing the name-lookup computation that would otherwise be needed. This thesis compares two different swizzling techniques using detailed timing studies of high-performance full-functionality implementations. The comparison shows that the two approaches have similar performance, so that the choice between them should be determined by other factors, such as simplicity of implementation.

1.1.2 Which objects are fetched into the cache

When there is a cache miss, obviously the needed object must be fetched; what else should be fetched? One possibility is to fetch nothing except the needed object; this is called a *single-object server*. Another possibility is to fetch a page containing the needed object and whatever else has been placed on the same page; this is called a *page server*. An earlier study by De Witt *et al.* [27] showed that a page server usually outperforms a single-object server. There is a third possibility, not considered by that study: it is possible to send a *group* of objects to the client. Since the size of an average object is on the order of 100 bytes, and a typical hardware page size is 4K bytes or 8K bytes, a page server is effectively sending a group of objects on each page. Rather than sending a group determined by the page size, it is possible to send groups determined by the connectivity of objects (and possibly other information). This thesis shows that sending such groups works better for varying workloads than an approximation of pages.

1.1.3 How objects are removed from the cache

To bring in new objects, the cache manager must find room for them. Some object(s) must be evicted from the cache, in a correct and efficient manner. This problem actually consists of several related problems:

- Which objects should be discarded?

- What happens to references to discarded objects (dangling references)?
- How can the freed storage be made available for allocation?

The thesis describes the techniques necessary for the system to perform correctly; it then goes on to describe a number of techniques for choosing objects to be discarded, and compares their performance on a simple benchmark. This is the first experimental work done on policies for discarding objects from a client cache. Previous work has been based on pages, or has dealt only with mechanisms and ignored policy.

The thesis describes two simple approaches to recovering client cache storage: garbage collection and *shrinking*. In shrinking, an object is replaced by a small placeholder data structure called a *surrogate*. The thesis demonstrates that neither technique is sufficient on its own, but that they do well when combined.

An appendix of the thesis presents a new algorithm, MINFETCH, for deciding which objects should be discarded from the cache for the hypothetical case when there is perfect knowledge of the future computation. The algorithm minimizes the number of fetches required. This problem is not solved by Belady's OPT algorithm [5] for paging systems, since OPT only deals with a single page at a time; the MINFETCH algorithm discards multiple objects of varying sizes. Since a paged system can only discard whole pages, MINFETCH can do better than Belady's OPT at minimizing fetches.

Finally, the thesis presents results from another set of experiments comparing various policies for discarding objects: MINFETCH and four different realistic online policies. An close approximation to a least-recently-used policy (called LRU) does very well, nearly matching MINFETCH; however, a coarser approximation of least-recently-used (called CLOCK) does little better than the other policies, which keep no usage information. An important related result is that for the experiments performed, varying the prefetching policy has a larger effect on system performance than varying the cache management policy.

1.2 Experimental Limitations

The conclusions of this thesis are based on experiments I ran with a prototype client-server object database called Thor[46]. Those experiments had two basic limitations. First, they were run with a particular synthetic workload. Second, they were run on a particular configuration of hardware and software. This section considers the effect of these limitations on the results.

1.2.1 Synthetic workload

The experiments use a part of the OO7 benchmark [12] for object databases. The intent is to capture the dominant effects of a generic computation in an object database. Accordingly, the experiments involve read-only traversal of inter-object references. Fast traversal of inter-object references is a key factor that motivates the use of object databases instead of relational databases [53]. Use of a read-only workload separates this fundamental component of performance from the question of concurrency control mechanisms and their performance. This separation is useful for understanding the nature of fetching, reference traversal, and cache management; but it also means that the results of this study must not be applied indiscriminately to environments with a large number of clients per server, especially if those clients perform transactions that frequently conflict. A limitation that the experiments inherited from OO7 is that the database is synthetic and contains objects of only a few different sizes.

Would one expect significantly different results with different kinds of computations? Consider first a computation with less pointer traversal. Decreasing the importance of pointer traversal decreases the importance of the results reported, but also decreases the value of an object database. Decreasing the importance of pointer traversal also makes it even less likely that an application would be able to tell the difference between different swizzling techniques, strengthening the conclusion that they are almost indistinguishable.

Now consider a computation that modifies objects. Allowing the mutation of objects can affect the cache management techniques described (details are in Chapter 6), but as long as the typical pattern holds of multiple objects read for each object written, there should be relatively little change in these results.

Finally, consider a computation with differently-sized objects. The number of different object sizes, and their arrangement in the database, was such that they did not fit conveniently into any uniform storage container. I would not expect significantly different results from a more varied collection of object sizes. The designers of the OO7 benchmark did not mention any concerns about the range of object sizes or the realism of the database in a recent status report [11].

1.2.2 Particular configuration

The experiments were run on a single machine (so there was no network delay) and in a way that ensured that all objects of interest were in memory (so there were no disk reads). Most of the results are applicable to other configurations, because neither swizzling nor shrinking policies are affected significantly by network delay or disk behavior.

However, the generality of one result is questionable because of the particular configuration used: the effectiveness of prefetching a dynamically-computed group. The results reported here assume that the objects of interest are effectively in the server's memory, meaning either that the objects really are in the server's memory or that it would be equally hard for object-fetching and page-fetching systems to get them off the disk. However, it is possible *both* that prefetching performance is critically dependent on scheduling disk accesses, *and* that a scheme based on pages does better at such scheduling, in which case the results showing the advantages of dynamic prefetching would not be correct. Regardless, the results remain valid for a server configuration in which the objects needed by the client are always in the server buffers (such as a main-memory database).

The configuration used does not write objects back to the server as part of cache management; writing objects back to the server may allow certain computations to proceed with smaller caches, at the potential expense of more complex bookkeeping and higher network traffic.

The limitations listed above actually strengthen some of the results. The omission of network delay and disk accesses reinforces the importance of prefetching as the primary determinant of performance.

1.3 The Rest of the Thesis

Chapters 2 and 3 describe the context of the work. Chapter 2 describes a model of a distributed object database, identifying the essential features of the system measured so as to clarify the applicability of the information presented in later chapters. Chapter 3 describes the database, and the traversals of that database, used for assessing performance. The database and traversals are adapted from the OO7 benchmark suite [12], which represents the best published benchmark suite for object-oriented databases. Chapter 3 also explains the constraints that led to using this approach for assessing performance.

Chapter 4 describes how inter-object references are represented at the client. The description begins with a simple system in which all objects are present at the client and inter-object references are implemented as memory pointers, so that an object A with a reference to an object B is implemented by having a field of A contain the address of B. Two different elaborations of this system yield the two different swizzling systems (called *node marking* and *edge marking*) that were implemented and measured. These swizzling systems are compared in terms of the performance of a simple *dense* traversal and a simple *sparse* traversal. Each such traversal is considered for three cases: when none of the objects used is present in the cache; when all of the objects are present but the garbage collector has not run; and when all of the objects are present and the garbage collector has run. These configurations represent extremes of program behavior; one would expect real programs to exhibit some combination of these behaviors. I conclude that edge marking holds a slight advantage in speed and space used in most cases, but that the difference between the two schemes is small enough that the choice can be decided based on other factors, such as simplicity of implementation. This result contrasts with some other published reports. Those reports appear to have been based on microbenchmarks that overemphasized the differences between swizzling techniques.

Chapter 5 describes the effect of several simple prefetching mechanisms on the same dense and sparse traversals. The prefetching mechanisms fall into five broad categories: breadth-first, depth-first, class-based, pseudopaging, and optimal. All except the optimal algorithm are *online* algorithms. All except pseudopaging choose an object group dynamically, at the time a fetch request is received. Several useful comparisons appear in the experimental results.

The first such comparison is between choosing object groups dynamically and computing them statically. On the dense traversal, the dynamic breadth-first prefetcher does nearly as well as a pseudopaging implementation optimized for the dense traversal; when both prefetchers are then used for a sparse traversal, the dynamic prefetcher does much better than the static prefetcher.

A second useful comparison is between prefetching techniques that use information about the likely uses of the objects and those that use only the structure of the objects at the server. The class-based prefetcher performs better than the prefetchers using only structural information, as long as its class-based information is fairly accurate. However, it performs worse than structural techniques if the database changes so that the information used is no longer accurate. Since structural techniques always use the current structure of the database, they do not have any similar problem with out-of-date information.

A third useful comparison is between techniques that use a server-side model of the client cache and those that do not use such a model. The results show that the model is important for performance: prefetching without the model runs 14% to 100% slower than prefetching with the model. Prefetching with the model in turn runs about 100% slower than perfect prefetching does.

Chapter 6 considers the mechanism for managing the client cache. The chapter describes a new mechanism that combines garbage collection with the *shrinking* of objects into surrogates. An experiment demonstrates that this combined mechanism allows the dense traversal (as used in previous chapters) to continue with much smaller caches than would be possible with either mechanism on its own.

Chapter 7 considers how to choose the objects that should be shrunk. One of the weaknesses of the OO7 benchmarks for this work is that there is no benchmark with a gradually shifting working set. So the chapter compares policies using a new *shifting* traversal, based on the dense traversal used earlier. The details of the traversal and database modifications also appear in Chapter 7. The chapter presents the MINFETCH algorithm (described earlier) and compares MINFETCH to four simple realistic policies. Those policies are TOTAL, which shrinks all objects in the client cache; RANDOM, which chooses objects to be shrunk at random from the collection of eligible objects; CLOCK, which implements a one-bit approximation of a least-recently-used policy; and LRU, which implements a true least-recently-used policy by stamping each object with a use count when it is used.

For the benchmark used, LRU comes close to the performance of MINFETCH. There is little difference in elapsed time between RANDOM and CLOCK, suggesting that one bit may be too coarse for eviction decisions at the granularity of a single object. There is little overall difference in elapsed time between TOTAL, CLOCK, and RANDOM, although TOTAL is less consistent in its behavior as the cache size changes. For applications behaving like the benchmark, the experiments suggest that the choice between these three policies should be based on implementation simplicity rather than arguments about performance. The experiments also show that choices about prefetching have a larger effect on performance than choices about client cache management.

Chapter 8 summarizes the content of the previous chapters, draws conclusions, and describes future work suggested by this thesis.

For each chapter where there is related work, a section near the end of the chapter discusses that work and its relationship to the content of the chapter.

Chapter 2

System Model

This thesis reports on experimental work done in the context of Thor [46], a distributed object database system currently being implemented at MIT. This chapter presents a model of a distributed object database that abstracts away many of the details of Thor.

The following features of Thor are crucial to the techniques described subsequently:

- There are functionally distinct clients and servers;
- Each server can distinguish reference fields from non-reference fields in its persistent objects;
- Objects at servers refer to each other using names, not memory addresses.
- Operations on objects are invoked via a runtime dispatch mechanism (this is one meaning of the much-abused term "object-oriented"); and
- Storage reclamation is implicit; i.e., there is some form of garbage collection in the implementation;

In addition to the features described above, the following feature is required to implement the *node marking* scheme that is described in Chapter 4:

- A static type system ensures that every field of every object has an associated type, and that no field ever contains a value that is not of that field's type.

As noted, the hardware structure of a distributed object database consists of clients and servers (see Figure 2-1). Persistent storage and its related functions are handled by servers, but operations of objects are executed in a portion of the database system that runs on client machines, as indicated by the partial inclusion of clients in the oval representing the database. Figure 2-2 shows a single client-server pair, along with a server disk. Both clients and servers use caches to avoid expensive accesses whenever possible. A client checks its local cache before requesting an object from a server. Similarly, a server checks in its cache before reading objects from disk.

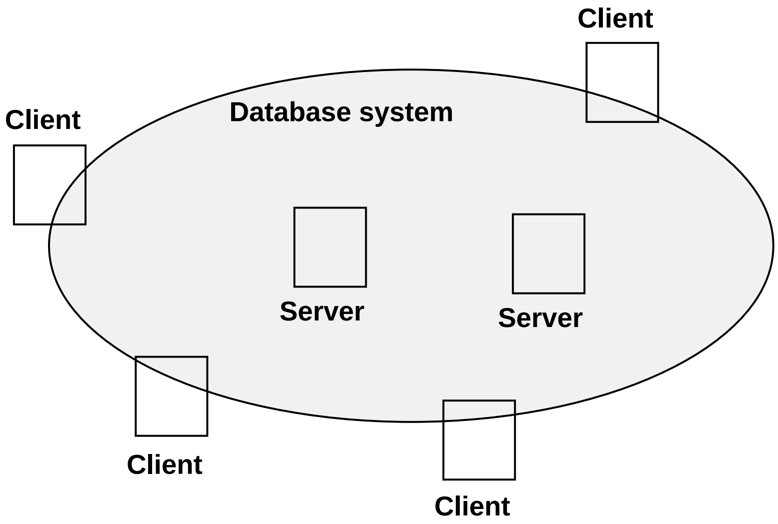

Figure 2-1: Clients and servers

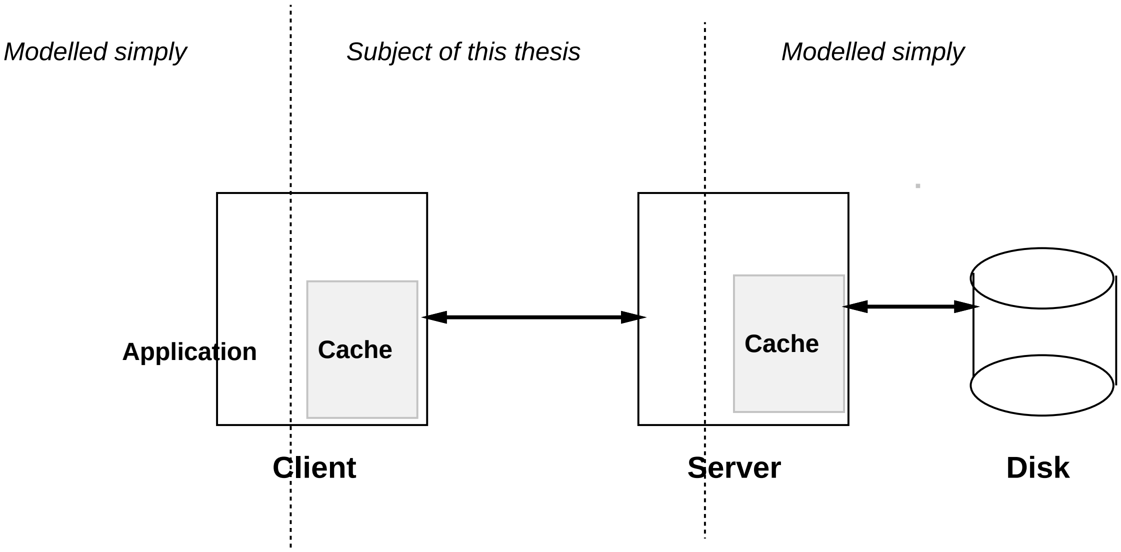

Figure 2-2: Client, server, and their caches

This thesis deals only with the management of the client cache. The experiments ensure that all objects of interest are in the server cache. The problems of managing the server cache and the server's disk are beyond the scope of this thesis; Ghemawat [29, 30] is addressing some of those problems.

This thesis also ignores the details of the application. Computations are described in terms of a database viewed as an *object graph.* The objects in the database are treated as nodes of a multigraph, and inter-object references are edges of that multigraph. Computational steps involving non-reference data are ignored; computations are treated as traversals of the object graph.

Chapter 3

Methodology

This thesis reports the results of a number of experiments with varying swizzling techniques, prefetching policies, and client cache management policies. This chapter describes the methods and data that are common to these experiments.

For these experiments, I built different versions of the Thor runtime system and drove those different versions with simple benchmark programs, described later in this chapter. At this writing, large portions of the Thor runtime system are complete, but it is difficult to implement new classes of database objects; the necessary compiler will not be finished for several months. In the absence of that compiler, I have built sample applications by hand-compiling class definitions into C, a laborious and error-prone process.

An alternate approach would be to use simulation, and indeed I used simulation for some early work. However, many of the problems of cache management depend on aspects of the system that do not have simple abstract models: for example, it is difficult to simulate storage management by garbage collection with much accuracy without building a mechanism quite similar to a real implementation. Further, in the absence of real applications, a simulation would still have to run the same synthetic benchmarks that I describe later; and the combination of a fake system with a fake workload seemed unlikely to yield any data of real interest. Accordingly, after deriving some early impressions from simulations, I have worked primarily on implementing and measuring the real Thor runtime system. The goal is to gain understanding of the system's likely behavior, not to exhaustively characterize the space of possible implementations. With this information in hand to resolve some early design questions, the implementation of the system can proceed; future work can re-examine the system's behavior with larger and more realistic applications.

The object graphs used are based on the database of the OO7 benchmarks [12] developed at the University of Wisconsin. The OO7 benchmarks are intended to capture some aspects of CAD applications, so as to show the consequences of implementation decisions in object-oriented databases.

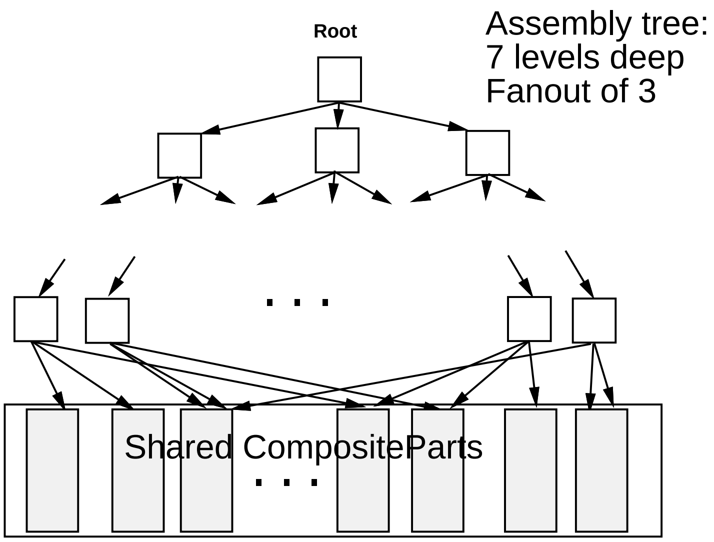

Figure 3-1: Structure of the OO7 Database

However, I am not aware of any study that has correlated performance on any part of OO7 with performance of a real application, so although the benchmarks make an effort to capture the behavior of applications, there is no proof that they are an adequate substitute for a real application. However, OO7 seems superior to other published synthetic benchmarks for object databases, such as OO1 [14] and HyperModel [3]. In a recent evaluation of lessons learned so far from OO7 [11], the authors do not indicate that they have received any complaints that their benchmark is particularly unrealistic or unrepresentative.

3.1 Database Structure

The OO7 database is made up of a tree of Assemblies with CompositeParts at the leaves (see Figure 3-1). Each CompositePart is in turn made up of a Document (2000 bytes of unstructured text) and a graph of AtomicParts and Connections (see Figure 3-2). Each Connection models a unidirectional connection from one AtomicPart to another. All of the entities shown — Assemblies, CompositeParts, AtomicParts, and Connections — have a number of data fields that are not shown, and many have other reference fields as well (for example, each Assembly also contains a reference back to its parent).

There are three traversals used in the thesis: dense, sparse, and shifting. All three traversals are "logically depth-first": when the traversal reaches an object with several children, the first child and all of its descendants are traversed before any subsequent children are touched. However, because of the extra reference fields, a "logically depth-first" traversal is not necessarily "physically depth-first"

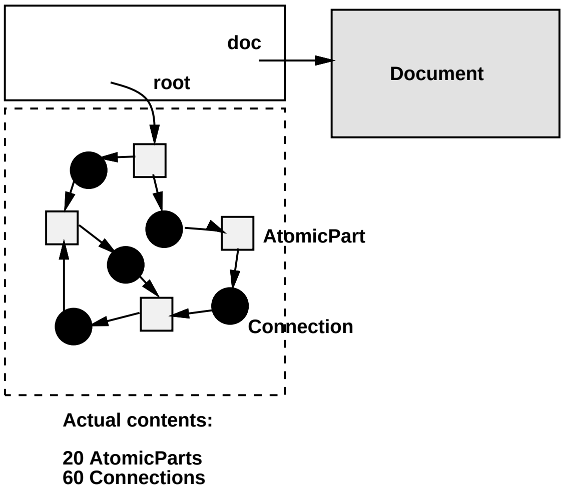

Figure 3-2: Structure of a OO7 CompositePart

in terms of the representation of the tree of Assemblies.

The dense traversal is based on OO7 traversal T1, and visits all Assemblies, CompositeParts, AtomicParts, and Connections. The sparse traversal is based on OO7 traversal T6 and visits all Assemblies and CompositeParts, but visits no Connections and only one AtomicPart per CompositePart. The shifting traversal (introduced in Chapter 7) is also based on OO7 traversal T1, but arranges a sequence of T1-like traversals so as to have a shifting working set.

Figure 3-3 shows the definition of DesignObj in the benchmark. Most of the other entities defined for the benchmark are also DesignObjs. This definition uses C++ syntax for the reader's convenience, although the actual Thor implementation is in an early dialect of the language Theta [22]. Client applications must manipulate objects entirely in terms of the operations provided by types; they have no knowledge of the fields of objects, nor any knowledge of the inheritance structure of classes. In terms of the definition provided, this means essentially that a client application can use only the public part of the interface, containing the methods.

Figure 3-4 shows the way in which benchmark types are related in terms of shared interfaces (their subtyping relationship). Any entity in the diagram is a subtype of any entity higher in the diagram to which it is connected. For example, a BaseAssembly can also behave like an Assembly or a DesignObj.

Figure 3-5 shows the way in which instances can contain instances of other types. The multiple levels of the tree correspond to the fact that a ComplexAssembly can contain other ComplexAssemblies.

Each traversal starts with getting the root object out of a Module. A Module both behaves like a DesignObj and inherits its implementation from DesignObj, which is defined in Figure 3-3. The

```
class DesignObj {
  public:
    int id();
    string otype();
    int buildDate();
    void set_buildDate(int);
  private:
    int id_;
    string otype_;
    int buildDate_;
  }
```

Figure 3-3: Definition of DesignObj

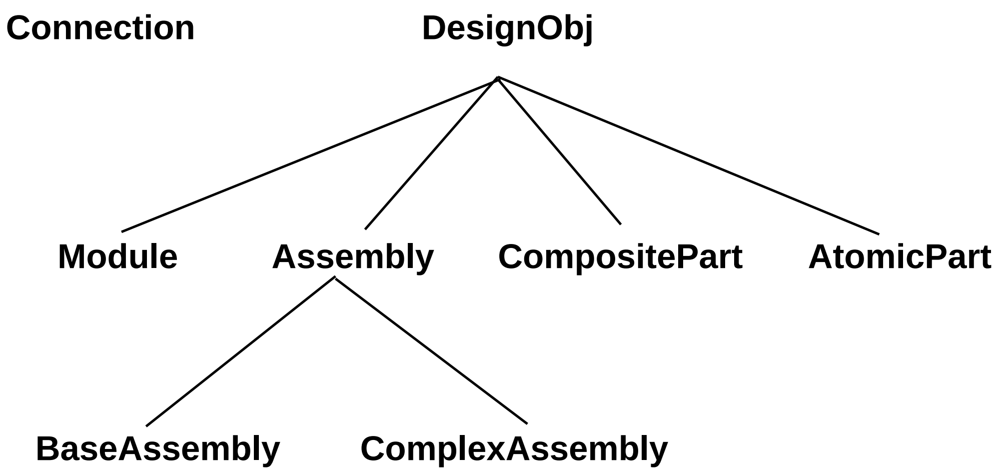

Figure 3-4: Subtype relationships among benchmark types

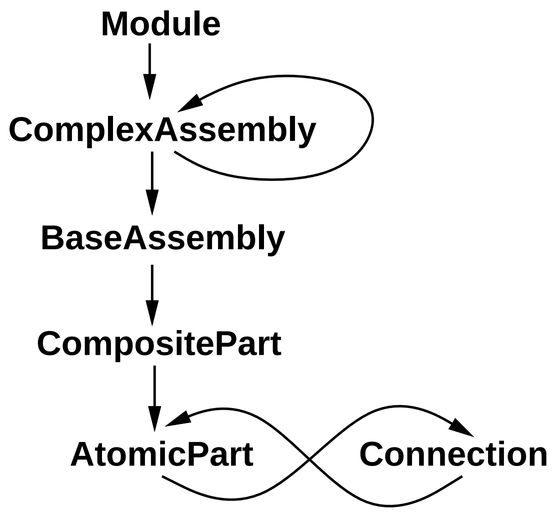

Figure 3-5: Containment relationships among benchmark types

fields of a Module, in addition to the fields defined by DesignObj, are called "man_", "assms", and "root". The only field of interest for traversals is "root", which contains the ComplexAssembly that is the root of the tree; the designRoot method returns the contents of this field.

There are two kinds of Assembly in the system: ComplexAssemblies and BaseAssemblies.

A ComplexAssembly has other Assemblies as children. In contrast, a BaseAssembly has CompositeParts as children.

A CompositePart consists of a number of AtomicParts, one of which is distinguished as the root AtomicPart.

In terms of the traversals, the only aspect of an AtomicPart of interest are the Connections that connect each AtomicPart to other AtomicParts. However, AtomicParts have a number of other fields intended to simulate the kind of data that might be kept for the smallest unit of a CAD application.

A Connection serves to connect one AtomicPart to another. This connection might need to have some kind of annotation or information in a CAD application, so a Connection has some fields of its own rather than simply being a pointer from one AtomicPart to another.

Figure 3-6 shows the sizes of the various kinds of objects in the database. The abstract size of each kind of object (in the second column of Figure 3-6) and the number of each kind of object (in the fourth column) are the same regardless of the details of how objects are implemented. However, the actual size of each object in bytes (shown in the third column) is determined by the sizes of the various primitive components, and those sizes are shown in Figure 3-7 for the current Thor implementation. The total database size is 41472 objects with a total size of 5569628 bytes, for an average object size of about 134 bytes.

part	size	bytes (each)	number	bytes (total)
ComplexAssembly	$h + 4f + v_3$	168	243	40824
BaseAssembly	$h + 5f + v_3$	176	729	128304
CompositePart	$h + 7f + v_1 + v_{20}$	500	81	40500
AtomicPart	$h + 7f + 2v_3$	272	10000	2720000
Connection	$h + 4f$	88	30000	2640000

Figure 3-6: Sizes of objects in database

variable	meaning	size (bytes)
h	Header and info overhead	56
f	Field	8
v_1	1-element vector	64
v_3	3-element vector	80
v_{20}	20-element vector	216

Figure 3-7: Sizes of object components

Since sets are not yet implemented in Thor, the database measured for this thesis uses vectors wherever the OO7 schema calls for sets. Accordingly, the results for Thor are not directly comparable to published OO7 results. The current Thor implementation of the database is probably slightly faster and smaller than it would be if it contained sets. However, the overall structure of the database, in terms of number of objects, relative sizes of objects, and their interrelationships, is quite similar to what it would be with sets instead of vectors.

The dense traversal is used in Chapters 4, 5, and 6. The sparse traversal is used in Chapters 4 and 5. The shifting traversal is used in Chapter 7. I defer further details about the databases and traversals to the chapters where they are used.

The traversals I use consist mostly of navigation from object to object. Accordingly, although these traversals attempt to capture aspects of performance that are important to applications, they should not be mistaken for realistic applications.

3.2 Discussion

The OO7 database is completely synthetic, and an intrinsic problem of any such benchmark is that it may not accurately capture the most important parts of the applications of interest. However, there are a number of positive aspects to using OO7 as a basis for these benchmarks. First and perhaps most important, it is small enough to be implemented fairly quickly while also being complex enough to capture some of the important aspects of performance in an object database. Using a recognized published benchmark as a starting point helps to keep the evaluation process honest; in particular it reduces the danger of inventing a test to make Thor look good. The OO7 benchmark captures some of the clustering and lumpiness of object data, and its parameters can be adjusted to construct plausible approximations of many different kinds of data.

Chapter 4

Swizzling

Objects stored at servers typically contain references to each other. Recall from Chapter 1 that an address is a "primitive" or "direct" reference, while a name involves some level(s) of interpretation. For the purposes of this thesis, a virtual memory address will be considered a direct address: that is, we will not be concerned with the levels of interpretation involved in dereferencing a virtual memory address.

Names are used for inter-object references in almost all persistent object systems, even those using pages. Using names instead of virtual addresses allows the data to exist independent of particular machines or machine architectures. In general, when an object is fetched from the server to the client, some of its references will be references to objects already present at the client, while the others are references to objects still at the server. A reference to an object cached at the client can be converted to use the address of the local copy of that object. Such a conversion is called *swizzling*. After swizzling, subsequent uses of the reference can avoid the overhead of translating the object reference to its current address. [39] was the first system to use swizzling; the technique has since become the norm for persistent object systems.

A system like Thor transfers object groups, not pages; Chapter 5 justifies that design choice. For this chapter the object-group-transfer architecture is simply assumed. In a system that transfers objects, there are two plausible granularities at which swizzling may take place: individual references or whole objects. Following Moss's terminology [52] I use *edge marking* as synonymous with swizzling at the granularity of individual references, and *node marking* as synonymous with swizzling whole objects. The terms come from considering objects (resp. references) to be nodes (resp. edges) of a directed graph, in which some entity must be "marked" to distinguish the swizzled form from the unswizzled form.

This chapter compares the performance of edge marking and node marking in Thor. The initial design of Thor [46] used a node-marking technique. Subsequently, I proposed an edge-marking technique. It was difficult to assess the importance of the swizzling technique in the absence of data from an implementation. There is no consensus in the literature: Moss [52] reasoned that node marking and edge marking should have roughly similar performance, but a later paper by Hosking and Moss [34] suggested in a footnote that edge marking is "not competitive" with node marking. Kemper and Kossmann's study [40] concluded that there were significant differences among swizzling techniques on different workloads. They went so far as to propose type-specific and context-specific approaches to swizzling, so that applications could mix swizzling techniques for performance. For example, an object might contain one infrequently-used reference implemented by edge marking (low space cost but some time cost), and a frequently-used reference implemented by node marking (higher space cost but lower time cost).

To settle the question, I implemented both node marking and edge marking and tested their performance. This chapter describes the two techniques, then compares their performance as implemented in Thor. That comparison shows that there is little difference between node marking and edge marking. The chapter then examines the details of the implementation, determining the costs of the elements that make up the two implementations and verifying that the result is reasonable. The last part of the chapter reviews related work.

4.1 Node Marking and Edge Marking

To explain the node marking and edge marking techniques that were implemented, this section considers one invocation on one object.

A Thor object in memory has the structure shown in Figure 4-1. The pointer to the object points to the object's method pointer. An object may have more than one method pointer, but that complication is not relevant to the issues that I describe here. The method pointer contains the address of a dispatch vector that contains addresses of code to implement methods. The dispatch vector is typically shared among many instances of the same class. Below the object's method pointer are fields that may contain either primitive data (such as integers) or references to other objects.

I consider the steps that are taken for one small part of a dense traversal (as described in Chapter 3). One step of that traversal is to determine the AtomicPart to which a Connection c is logically connected. The invocation "c.to()" returns this AtomicPart. In the current implementation of Connection, the object c has a field (called the "to field" hereafter) that contains a reference to this AtomicPart, and the code that runs in response to the invocation "c.to()" simply returns the contents of this field.

In the proposed standard terminology for object systems [65], the invocation "c.to()" is a *request*. It identifies an operation on the object c, but does not determine how the operation will be performed.

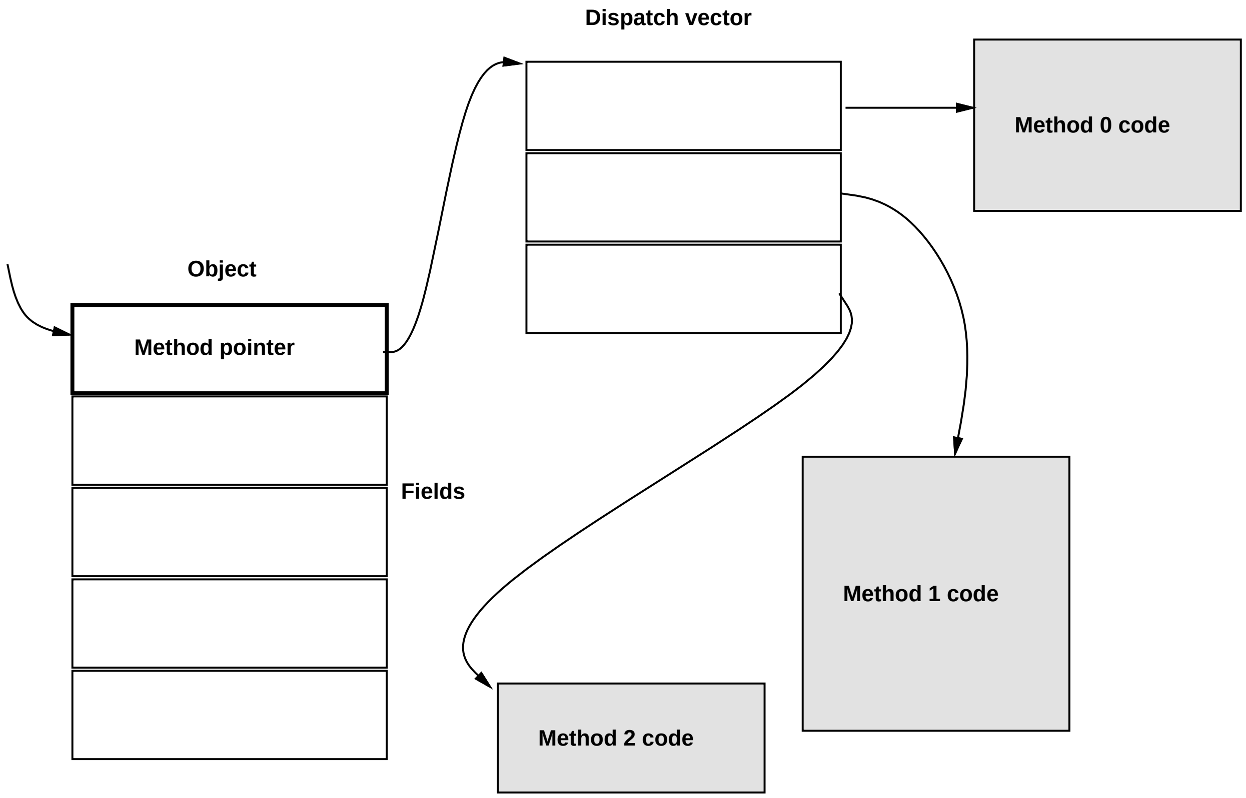

Figure 4-1: A Thor object, its dispatch vector, and methods

The code that actually runs in response to a request is called a *method.* The method to be executed is determined by a runtime dispatch using the object's method pointer.

Executing the invocation involves the following abstract steps:

1. *Method-finding:* Find the method implementing c.to.

2. *Computation:* Run that method, returning the result AtomicPart.

Slightly different steps are taken to implement these two steps, depending on the swizzling being used. The next subsections consider the details of those steps for a client-only system (no fetching and swizzling involved), a system using node marking, and a system using edge marking.

4.1.1 A Client-Only System

This section describes the steps that would be required to find the method implementing c.to and run the method in a simple client-only system, with no fetching from a server. In such a system, all of the objects are present in memory, and all inter-object references are simply memory pointers. In such a system, method-finding for "c.to()" involves only the following steps:

1. Load c's method pointer.

2. Use an indexed load to find the address of the method to be run as the implementation of the "to" request.

The computation involves only the following steps:

1. Jump to the method, passing c as "self".

2. Method returns the value of the "to" field.

4.1.2 An Edge-Marking System

Now consider what happens for the same invocation when using edge marking. The method-finding is identical to that described for the system that does not fetch objects (Section 4.1.1). However, the computation is different from what was described for that system:

1. Jump to the method, passing c as "self".

2. Method tests the pointer stored in the "to" field.

3. If it does not point to a resident object (i.e., it contains a name X),

    (a) Use the name X to look up an address q in the Resident Object Table.

(b) If there is no such address, the object-fetching routine is called, passing it the object name X. The object-fetching routine carries out a number of steps (including updating the ROT), explained later in section 4.2. It returns the address q of the local copy of the object.

(c) Put the object's address q into the "to" field, replacing its name X.

4. Return the value of the "to" field.

The *Resident Object Table* mentioned above maps object names to addresses for objects that are present in the client cache, and is used in a similar fashion for all object-fetching systems.

Before proceeding to node marking, consider the implications of and alternatives to this edge marking technique. As presented, the "to" field is swizzled (converted to a pointer to the fetched object) even though that object may not be used subsequently. Although the method call returns the "to" field to the caller, the computation may actually not use that object. The technique described is called *swizzling on discovery*, in contrast to *swizzling on reference*, which would wait until using the reference to test it and possibly fetch it.

Swizzling on discovery assumes that any object that is accessed is highly likely to be used, in the sense of having some request invoked on it. Swizzling on reference, in contrast, assumes that it is advantageous to wait until the last possible point to swizzle a reference. Thus, in swizzling on discovery, one tests the instance variable of an object to ensure that it is swizzled, before allowing that (swizzled) value to be returned from methods. However, variables containing the results of such method calls can be used without needing any such test. In contrast, swizzling on reference means that the value of an instance variable can be returned without testing, but the first use of a variable containing such a value requires a test to ensure that the reference has been swizzled.

Swizzling on discovery can cause unneeded objects to be fetched: as soon as an object field is accessed, the corresponding object is fetched, even though subsequent computation may not actually use that object. For example, even if two fields could be compared for equality by comparing their names, swizzling on discovery will cause the object(s) referenced by the fields to be fetched. Swizzling on discovery ensures that every reference used has been swizzled in the cached object. Swizzling on reference avoids unnecessary fetches but often fails to gain any significant benefit from swizzling. How, then, should one choose between these alternatives? White and DeWitt have shown [73] that computations often fetch a reference from an object into a local variable before using that reference. This is true even for implementations of methods. Swizzling on reference can only swizzle the location where the reference is when it is used; if that location is a local variable, swizzling on reference swizzles only that local variable. At that point in the computation, the swizzling mechanism has no way of translating the field of the object from which the reference was fetched. Accordingly, swizzling on reference provides little gain unless that particular local variable is used repeatedly before being discarded at exit from its stack frame.

4.1.3 A Node-Marking System

Now consider the steps that occur if the system uses node marking and object c is not present. The described variant of node marking allows the system to run at full speed, without residency checks, under certain circumstances. An object that is referenced by a swizzled object but is not present in the cache is represented by a *surrogate* at the point when the request c.to() takes place. A surrogate is a small data structure (containing only one field) that acts as a placeholder for a real object until it is fetched. When a computation attempts to invoke a method of such a surrogate, the corresponding object is fetched from the server, the surrogate is modified to point to the newly-fetched object, and the corresponding method is invoked on that object. To support this behavior, a surrogate has two states: empty and full. If a surrogate is *empty*, its corresponding object has not been fetched, and the surrogate's field contains the name of the object; if the surrogate is *full*, the corresponding object has been fetched, and the surrogate's field contains the address of the object in the cache.

There are three cases to consider for the request "c.to()":

1. The object c is absent from the cache, and therefore is represented by an empty surrogate.
2. The object c is present in the cache.
3. The object c is present in the cache, but the invocation takes place on a full surrogate.

I consider each of these cases in turn.

First assume that the object is represented by an empty surrogate. Then each "method" of the apparent object c is an instance of a special piece of code called FETCH. The effect of FETCH is to fetch the corresponding object from the server, then invoke the method originally invoked on the surrogate. There is a different FETCH_i for each method index i; this is necessary so that it is possible to determine which method should be invoked after the right object has been fetched into the cache.

When FETCH_i runs, the following occurs:

1. The name of the object, X, is loaded from the surrogate's field.
2. When there is single-object fetching, it is guaranteed that the object corresponding to the surrogate is not present at the client. However, in general, prefetching may have occurred (as described in Chapter 5), so the following two steps are necessary to avoid redundant fetching:
   (a) The name X is looked up in the Resident Object Table (ROT).
   (b) If there is an entry for X in the ROT, it contains an address q for the object.
3. If the system is doing single-object fetching, or if the check of the ROT showed no entry for the name X,

(a) The object-fetching routine is called, passing it the object name X. The object-fetching routine carries out a number of steps (including updating the ROT and creating new surrogates as needed), explained later in section 4.2. It returns the address q of the local copy of the object.

4. Address q is stored into the surrogate's field, overwriting the object name X previously stored there

5. The surrogate's method pointer is changed to point to a dispatch vector of the same length, containing only pointers to FORWARD$_i$. (The function of FORWARD is explained below).

6. The method pointer of q is loaded.

7. An indexed load, using i as the index, finds the address of the method that implements the request originally made.

Now consider the steps that occur if object c is represented by a full surrogate. The method lookup proceeds as usual, but since the "object" used is a full surrogate, each method is an instance of a special piece of code called FORWARD. As with FETCH, there is a different FORWARD$_i$ for each method index i. When FORWARD$_i$ runs, the following occurs:

1. The address q of the real object is loaded from the surrogate's field.

2. The method pointer of q is loaded.

3. An indexed load, using i as the index, finds the address of the method that implements the request originally made.

The execution of the method ultimately found proceeds as for a system that does not fetch objects.

Finally, it is possible that the object c is present in the cache and is actually being invoked directly (rather than through one or more surrogates). In such a situation, the computation of a request such as c.to() is indistinguishable from the computation as originally described for a system that does not fetch objects. Thus, after fetching objects to the client and running the garbage collector, a subsequent computation on the cached objects incurs no overhead that is attributable to the object-fetching mechanism. This lack of overhead contrasts with edge marking, where every use of an object field requires a test even after all of the objects used are present in the cache. The same lack of overhead is the main motivation for considering node marking, since it is both more complicated and more expensive in space than edge marking.

4.1.4 Discussion

A full surrogate is logically unnecessary: instead of filling the surrogate, it would be possible to find all pointers to the surrogate and change those so that they point to the real object instead. However, there are only two ways to find all of the pointers to a particular surrogate: one is to trace all of the pointers from the roots, and the other is to keep a back-pointer to every object that refers to a surrogate. Both techniques are expensive. The tracing of pointers from roots is already a part of the garbage collector; rather than eliminating any surrogate on each fetch, full surrogates are left in place until the garbage collector runs, and the garbage collector eliminates the full surrogates. On encountering a full surrogate, the garbage collector updates the object pointing to the surrogate so that it points directly to the object reachable through the surrogate, bypassing the full surrogate. After the garbage collector has updated all of the pointers to the full surrogate in this way, the full surrogate is no longer reachable and is reclaimed like any other garbage.

4.2 Fetching

This section describes the details of an object fetch, which occurs when needed for both the edge-marking and node-marking systems.

4.2.1 Fetching with Edge Marking

In edge marking, fetching an object requires the following steps:

1. The server is asked to send the object named X

2. The server sends the object

3. The object is installed in the client cache by entering its name X and local address q in the Resident Object Table (ROT) that maps object names to addresses

4. The address q of the newly-installed object is returned to the caller of the fetch routine.

4.2.2 Fetching with Node Marking

In node marking, the first three steps are identical to those described above for edge marking. However, there is more work to be done after installing the object in the client cache, so the following steps take place:

4. The object is swizzled: for each reference contained in the object, the object name is looked up in the ROT.

   (a) If there is an address for that name in the ROT, that address is used to overwrite the name stored in the object's field.

   (b) Otherwise (there is no entry for that name in the ROT), a new surrogate is created that contains the name:

   i. The type of the surrogate is determined by the known static type of the field being swizzled.
   ii. The surrogate type determines how long the surrogate's dispatch vector must be.
   iii. The object's name is written into the one data field of the surrogate.
   iv. The address of the new surrogate is entered in the ROT, and used to overwrite the name in the object being swizzled.

5. The address q of the newly-installed object is returned to the caller of the fetch routine.

The apparent type of the object is known when an empty surrogate is created. This is the one point at which the system requires that the computations be in a strongly-typed framework. Here is an example of how this type information is used. Assume that an object y is fetched into the cache and swizzled, and y contains a field of type T referring to object z not in the cache. Type T is used when creating a surrogate for z. Type checking ensures that even if z has an actual type S that is a subtype of its apparent type T, the surrogate for z will not be asked to respond to a request that is allowed by S but not allowed by T. The apparent type (T in the example) determines how many methods the real object (apparently) has, and accordingly how many FETCH_i's there must be in the empty surrogate's dispatch vector.

Creating surrogates is an expense that was not needed for edge marking. In addition, there is more swizzling taking place, which may not be worthwhile. The fields swizzled may not be used by the invoked method triggering the fetch or any method subsequently invoked by the client application, in which case the effort spent swizzling them is wasted.

4.3 Apparatus

I implemented both node marking (with surrogates) and edge marking in Thor. The two mechanisms are implemented using conditional compilation, so that there is no runtime cost or code space associated with the system's ability to swizzle in two different ways. Below I describe the object graph used, the traversals of the object graph, and the nature of the control transfer between the test application and Thor. See Chapter 3 for an overview of the object graphs and traversals.

4.3.1 Object Graph

The object graph used for these experiments is essentially identical to the database called "small" in OO7. The difference, as mentioned earlier, is that OO7 specifies sets in numerous places; the database used in this thesis has vectors instead, since sets are not yet implemented in Thor. This database is small enough so that both the server and the client can keep the whole database in memory, without needing to use virtual memory or explicit cache management at the server.

Parameter	Value	Explanation
NumAtomicPerComp	20	AtomicParts per CompositePart
NumConnPerAtomic	3	Connections per AtomicPart
DocumentSize	2000	Bytes in each CompositePart's Document
NumCompPerModule	500	Total number of CompositeParts
NumAssmPerAssm	3	Fanout of Assembly tree
NumAssmLevels	7	Height of Assembly tree
NumCompPerAssm	3	Fanout at leaves of Assembly tree

Figure 4-2: OO7 Parameters for the database used

The database is generated by first building a library of 500 CompositeParts, then building the Assembly tree from the root down and choosing CompositeParts at random from the the library as needed. There are 3^7 or 2187 CompositeParts used in the database, so a CompositePart has 4 parents on average. Each CompositePart contains 20 AtomicParts. Each AtomicPart has 3 Connections to other AtomicParts. Accordingly, each CompositePart contains 60 Connections.

4.3.2 Traversals

There are two relevant traversals from OO7, one sparse and one dense. The dense traversal does a depth-first search of the Assembly tree; for each CompositePart encountered, the traversal does a depth-first search of the graph of AtomicParts and Connections contained in the CompositePart. The sparse traversal is essentially a truncated form of the dense traversal. Like the dense traversal, the sparse traversal touches all of the Assemblies and CompositeParts. However, it does not examine the internal structure of a CompositePart.

4.3.3 Measurements

The client and server ran as separate processes on a single DEC 3000/400 workstation. Times were measured with the processor's built-in cycle counter. The times measured were elapsed wall-clock times for sparse and dense traversals of the database, in three different cache states. A *cold* cache contains none of the objects used by the traversal, so that every object used must be fetched from the server. The other two cache states are both *hot*: the traversal starts with all of the objects needed in the cache. One traversal, *hot with garbage collection*, runs a traversal after a garbage collection; the other, *hot without garbage collection*, runs a traversal when all of the objects are present but the garbage collector has been prevented from running. As described previously, the garbage collector serves to remove a level of indirection when using node marking.

4.3.4 Transfer of Control

These traversals run almost entirely inside Thor; after an initial call from the client application, the rest of the traversal takes place inside the database. This is the structure that we might expect to see for the portions of an application that are performance-critical. Putting the application inside Thor ensures that the experiments measure cache management costs, rather than the cost of crossing the protection boundary separating application from database.

4.4 Hypotheses

I propose and test three hypotheses concerning node marking and edge marking.

1. *Edge marking is faster than node marking on the cold traversals.*

   I expected that edge marking would perform better on a cold traversal, since it avoids creating and then filling in the surrogates that are used by node marking.

   This hypothesis is supported by my experiments, but edge marking's advantage is small, since the cost of fetching dominates.

2. *Edge marking is faster than node marking on the hot traversals without garbage collection.*

   I expected that the node-marking implementation would perform quite poorly on the hot traversal if the garbage collector were not able to snap out the full surrogates. In such circumstances, the node-marking implementation suffers an extra indirection whenever it accesses an object through a full surrogate, as well as executing a small amount of extra "forwarding" code in the full surrogate. In the same circumstances, an edge-marking implementation must check each reference before using it, but does not encounter any other problem.

   This hypothesis is supported by my experiments; edge marking is 30–40% faster than node marking in this case.

3. *Node marking is faster than edge marking on the hot traversal with garbage collection.*

   I expected that node marking would perform better than edge marking on a hot traversal after a garbage collection, since the node marking code can run at full speed, as though it were a memory-resident system with no overhead due to fetching. The node marking code runs without checking pointers, while the edge marking code must check each pointer in an instance variable before dereferencing it.

   This hypothesis is supported by my experiments, but node marking's advantage is quite small.

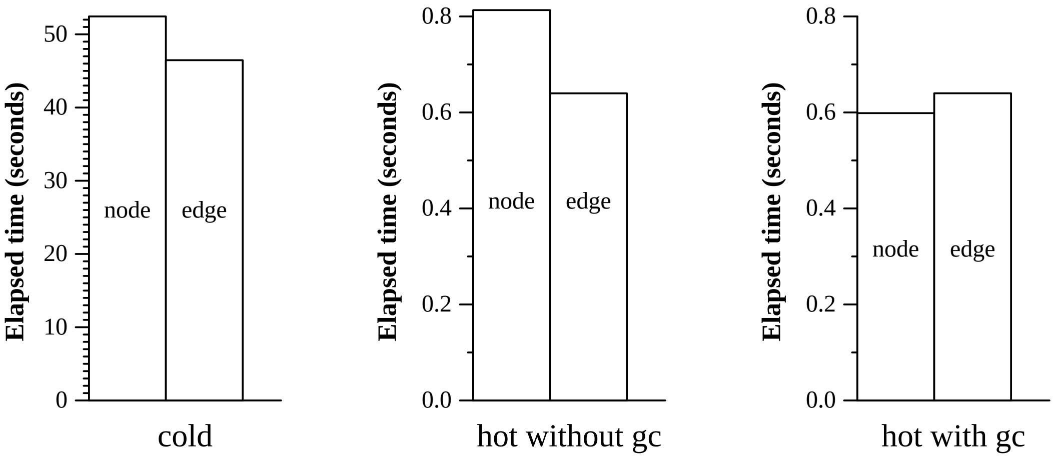

Figure 4-3: Node marking vs. edge marking, dense traversal

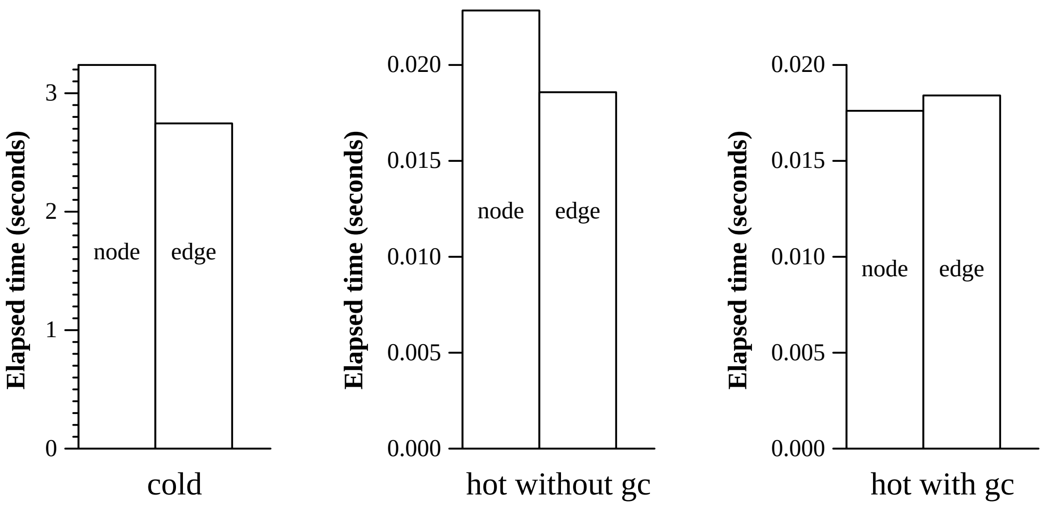

Figure 4-4: Node marking vs. edge marking, sparse traversal

4.5 Results

Figure 4-3 compares edge marking and node marking for the dense traversal in the three cache states: cold, hot without gc, and hot with gc. Figure 4-4 similarly compares edge marking and node marking for the sparse traversal.

As expected, edge marking does slightly better than node marking for the cold traversal, and node marking does slightly better than edge marking for the hot traversal after full surrogates have been eliminated. Edge marking does markedly better than node marking for a hot traversal where surrogates have not been eliminated. The current Thor garbage collector is triggered either when space runs low or when the rate of storage allocation is unsustainable. Since each traversal runs as a single operation, the cold traversal appears to be a single operation that allocates an enormous amount of storage, and so the garbage collector typically runs after the cold traversal. Accordingly, special code is required so that Thor will stay in the "hot without gc" state for these measurements.

The garbage collector reported here is actually a special garbage collector that only snaps out surrogates. With the real copying garbage collector, the first garbage collection causes a slight decrease in performance, even for edge marking. This effect occurs only at the first garbage collection, and appears to be related to changes in the (hardware) cache behavior when objects are rearranged in memory. For the purpose of understanding swizzling, this effect is irrelevant. For the garbage collector that only snaps out surrogates but does not rearrange or compact storage, edge marking performs identically before and after a garbage collection.

A real application is likely to be some mixture of these three regimes, and so it seems that edge marking is likely to be somewhat superior to node marking. However, the difference between edge marking and node marking does not seem large enough to merit special attention from the programmer.

4.5.1 Reading code

I examined the code that implements the various primitive operations described in an attempt to determine their relative costs. For all of these measures, I am describing code generated by the DEC cxx compiler (C++ for OSF/1 AXP systems v1.2) running with optimization -O2 on a DEC 3000. I follow Myers's model [55], assigning a cost of L to each load that is unlikely to hit in the hardware cache, a cost of J to each indirect or conditional jump instruction, and a cost of 1 to each other instruction. The instructions with a cost of L or J are so marked on the left margin.

Figure 4-5 shows the code for a full surrogate (what I have been calling FORWARD_i). The expensive loads and jump are marked in the left margin of the code. The first three lines "bump" the original pointer to self in a0; this is a step that is necessary only when a method must actually access fields of an object (as it must in this case). The subsequent lines fetch the address of the real

```
        # This is hand-written assembly code

(L)     ldq     t1, 0(a0)           # load method pointer
        ldq     t2, 8(t1)           # load offset
        addq    a0, t2, t3          # add offset
(L)     ldq     a0, 0(t3)           # load new "self"
(L)     ldq     t1, 0(a0)           # load new method pointer
        ldq     pv, 32+i*8(t1)      # look up real method
(J)     jmp     (pv)                # jump indirect to that method
```

Figure 4-5: The code for FORWARD_i

object that should be used as self and make the appropriate arrangements to jump to that object's methods. This code has a cost of $3L + J + 1$. This code has been written directly in assembler rather than being generated by the compiler.

Figure 4-6 shows the code for testing a pointer in the edge marking scheme. When the referenced object does not need to be fetched, the code requires only a single additional jump, for a cost of J. When the object needs to be fetched, the cost is $2J + 6$ since most of the loads and stores are of entities likely to be in the cache; however, this overhead is not of much interest since the cost is swamped by the much larger cost of a fetch from the OR.

On a processor like the DEC Alpha, it would be possible to reduce the test cost to 1 (instead of J) for some simple methods. The Alpha predicts that forward branches are not taken, while backward branches are taken. Figure 4-7 shows how the code of Figure 4-6 could be rearranged. This assumes that the code following label $45 ends with a return to the method's caller, as shown. If that code is placed immediately *after* the test of the loaded pointer, and the sense of the test is inverted, the branch prediction hardware works correctly. This optimization must be done separately for each pointer test, so it is only appealing if one has control of the compiler. Since I generated machine code for most methods using the C++ compiler supplied by Digital, I did not implement this further optimization. However, it is worth keeping in mind when comparing the performance of edge marking and node marking.

4.5.2 Measured costs

I compare these instruction counts to the costs that I could determine from measuring the running system. I used the on-chip cycle counter to measure these times on a DEC 3000/400, which has a clock of 133 MHz for a cycle time of 7.5 ns.

```
        # "self" is in t0 when this code starts running
        # This code is generated by the C compiler

        # This load has to happen anyway
        ldq     a0, 72(t0)             # load pointer to be tested

        # This is the pointer-testing code
(J)     bge     a0, $45                # test
        bis     zero, zero, a1         # pass default hint (0)
        stq     t0, 16(sp)             # save self
(J)     jsr     ra, cache_fetch_pointer
        ldgp    gp, 0(ra)              # restore global pointer
        ldq     t0, 16(sp)             # restore self
        stq     v0, 72(t0)             # store back swizzled value
        ldq     a0, 72(t0)             # use swizzled value
$45:                                   # continue with rest of method
```

Figure 4-6: The code for testing a pointer in the edge marking scheme

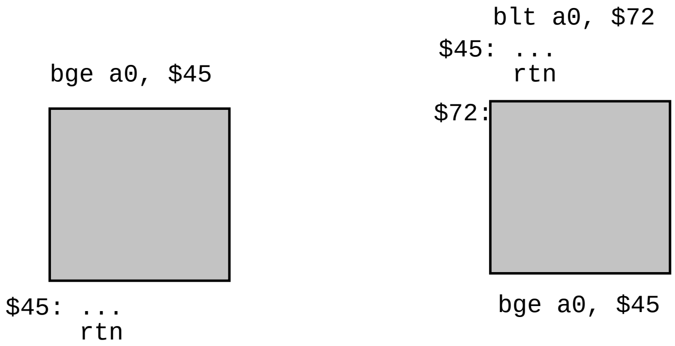

Figure 4-7: Rearranging the pointer test for higher performance

The cost of testing a pointer

The first measurement determines the cost of testing a pointer in the edge marking scheme, and thereby approximately determines the value of J. I measured the times for two hot traversals with garbage collection: one was implemented using node marking, and the other one was implemented using edge marking. The difference between these times, divided by the number of pointer tests, gives the cost of testing the pointers. The relevant results for the dense traversal are:

41 ms
450545 pointer tests
$=$ 92 ns / test
$\approx$ 12 cycles / test

This implies that $J \approx 12$.

The cost of FORWARD_i

The next measurement is the cost of using a full surrogate (running FORWARD_i). For a node marking system, the difference between a hot traversal without garbage collection and a hot traversal with garbage collection is exactly the cost of going through a full surrogate for each object. So the difference between the two traversal times, divided by the number of object requests, gives the cost of running FORWARD_i. The relevant results for the dense traversal are:

215 ms
532425 surrogates used
$=$ 400 ns / surrogate used
$\approx$ 53 cycles / FORWARD_i

Since we know that FORWARD_i has a cost of $3L + J + 1$, this gives a value of $L \approx 13$.

So far, the measurements suggest that the values for our model are $L = 13$ and $J = 12$. These numbers are rather too high for the processor used [54], suggesting that some of the instructions assigned a unit cost by this model in fact had a higher cost.

The cost of fetching

I determined the cost of fetching objects by running a cold edge marking traversal and comparing that to a hot edge marking traversal without garbage collection. (Recall that these cold traversals also do not have garbage collection.) The difference between the times, divided by the number of fetches, represents the average cost to fetch an object. The relevant results for the dense traversal are:

Operation	Instructions counted	Cycles measured
Check pointer	J	12
Use full surrogate	$3L + J + 1$	53
Swizzle object		8300
Fetch and install object		117,000

Figure 4-8: Instruction counts vs. Measured times

45.8 sec
51979 fetches
$= 882\ \mu$s / fetch
$\approx$ 117,000 cycles / fetch

The cost of surrogate creation

Finally, I determined the overhead of creating surrogates and swizzling objects by running a "cold" node marking traversal. I then subtracted out the costs previously computed for fetching the objects and traversing the objects (and surrogates). The resulting time, divided by the number of surrogates created, represents the cost of swizzling and creating a surrogate. The relevant results for the dense traversal are

5.80 s
93394 surrogates created
169098 references swizzled
$= 62\ \mu$s / surrogate created
$\approx$ 8300 cycles / surrogate created

These are somewhat unsatisfactory numbers, since they mix together the costs of swizzling and surrogate creation. In addition, there is no way within this particular experiment to distinguish the cost of using an empty surrogate from the cost of using a full surrogate, even though we know that they have quite different costs.

4.6 Code Expansion Effects

A performance penalty not measured by my experiments is the effect of code expansion on the cache hit rate. In this section I explain this potential problem.

In the edge marking scheme, every piece of code that fetches an object reference from a field must test that reference before proceeding. This test means that a method body that requires only a single instruction in a client-only system may require a dozen instructions in an edge-marking system, and this is true for every method of every class. There is no comparable code size penalty for the node-marking scheme, since the effect of the test is accomplished by executing the shared FETCH_i code when needed. Depending on the workload, there are three possible cases for the code size in relation to the cache size:

1. The code working set is much smaller than the hardware cache, so that the set still fits in the hardware cache even if the code expands.

2. The code working set is only slightly smaller than the hardware cache, so that the expansion changes the behavior of the system from mostly hitting in the hardware cache to frequently missing.

3. The code working set is larger than the hardware cache, so that the system is already getting little value from the cache.

Only in the second case is the code expansion from edge marking a significant performance issue.

The code expands the most, as indicated previously, for very simple methods that only access single fields. Any other code in the system, whether for accessing non-object fields, updating fields, or computing on local variables, tends to dilute the difference between node marking and edge marking. In addition, each field access method of a mutable object must record information about the transaction status of the object; since this code must be present in both edge marking and node marking systems, it also tends to dilute the code expansion effect.

Although edge marking performed well in the experiments I ran, there is too little code in my benchmarks to determine the true effect of code expansion on instruction cache performance. Currently, it is too hard to generate code to perform realistic experiments on code expansion and its effects (if any). Useful future work would include an examination of real applications running on Thor to determine their instruction cache miss rates. If it appears that performance could be significantly improved with smaller code, it may be worth revisiting the node marking scheme.

4.7 Related work

As previously indicated, Moss [52] first categorized swizzling techniques as *edge marking* or *node marking*. Kemper and Kossmann [40] subsequently distinguished techniques based on their eagerness (eager vs. lazy) and directness (direct vs. indirect). I start from Kemper and Kossmann's terms to categorize related work on swizzling techniques.

There are two relatively independent dimensions of swizzling techniques. The first dimension is *directness* (direct, indirect, or mixed) and the second dimension is *eagerness* (eager or lazy).

	Eager	Lazy
Indirect	LOOM [39], others	LIS [40]
Mixed	Thor node marking	Thor edge marking with surrogates
	Persistent Smalltalk [33]	
Direct	O_2 resident mode [4]	Thor edge marking
		Exodus [73]

Figure 4-9: Swizzling in different systems

Consider directness first. A *direct swizzling* technique translates an inter-object reference into the address of the object itself. Edge marking in Thor, as described in this chapter, is an example of direct swizzling. Similarly, Exodus [73] uses direct swizzling, as does O_2 [4] in its so-called resident mode. In contrast, an *indirect swizzling* technique translates such a reference into the address of a descriptor for the object, which in turn contains the address of the object itself. Direct swizzling is more efficient than indirect swizzling in both space and time. However, indirect swizzling simplifies storage management because object descriptors make it easy to relocate or evict objects. Examples of systems using indirect swizzling are LOOM [39], Emerald [38], Orion [41] and Jasmine [36]. Some systems have *mixed swizzling*, in which an inter-object reference may be direct at some times and indirect at others; these systems are attempting to get some of the good properties of both direct and indirect swizzling. Node marking as described in this chapter is an example of mixed swizzling: inter-object references via full surrogates are indirect, but those full surrogates are snapped out by the garbage collector and replaced by direct pointers. Persistent Smalltalk [33, 34] is another example of mixed swizzling; but instead of using the garbage collector to snap out the descriptors (which are called indirect blocks in Persistent Smalltalk), there is a special scanning step that runs as part of an object fault. The scanning finds indirect references and replaces them with direct pointers.

Now consider eagerness. An *eager swizzling* technique ensures that any object in the client cache contains only swizzled pointers; there are no names left in object fields. Node marking as described in this chapter is an example of eager swizzling. The resident mode of O_2 [4] is another example of eager swizzling. In contrast, a *lazy swizzling* technique allows object fields to contain names or swizzled pointers, converting names to pointers only as needed. Edge marking as described in this chapter is an example of a lazy swizzling technique; Exodus [73] is another example of a lazy swizzling system.

We can combine these two dimensions into the table of Figure 4-9, which shows where various systems fit in. LIS, which is shown as an example of lazy indirect swizzling, is not actually a system. Instead, it refers to a study done by Kemper and Kossmann on the four combinations of lazy/eager and direct/indirect. As far as I know, no actual system has used lazy indirect swizzling, since it requires both a test *and* an indirection to implement each inter-object reference. The entry for "Thor edge marking with surrogates" refers to a system that will be used in Chapter 7. For a system that only fetches objects, as in this chapter, it would be redundant to have surrogates as well as edge marking. However, when objects are to be evicted from the cache, surrogates can be useful even if fetched objects are swizzled using edge marking.

I am aware of three groups that have done swizzling studies, all using databases derived from the OO1 benchmark [14]: White and DeWitt [73], Hosking and Moss [33, 34], and Kemper and Kossmann [40].

White and DeWitt compared different implementations of lazy direct swizzling (edge marking) and swizzling at the granularity of pages (ObjectStore [42]). They found that swizzling on discovery is superior to swizzling on reference (Section 4.1.2 explains the difference). They also showed that a software swizzling scheme using edge marking is competitive with ObjectStore, even though ObjectStore takes advantage of virtual memory hardware.

Although an early analysis by Moss [52] matched my results, my results appear to differ from the more recent report of Hosking and Moss, who found edge marking "clearly uncompetitive" ([34], footnote 7). However, a personal communication [35] describing their implementation in more detail explains that their edge marking scheme is not swizzling: instead, it is doing repeated lookups, and accordingly is not comparable to the edge marking described here.

Kemper and Kossmann's study [40] concluded that there were significant differences among swizzling techniques on different workloads. Kemper and Kossmann went so far as to propose type-specific and context-specific approaches to swizzling, so that an object containing two references might have one reference translated by lazy direct swizzling, while another reference might be translated by eager indirect swizzling. Kemper and Kossmann's measurements focused on micro-benchmarks, such as determining the time to traverse a single reference; I believe this emphasis may have misled them as to the importance of swizzling compared to prefetching and other factors in the system. While it is possible to measure large differences in the cost of some primitive operations, those operations do not necessarily dominate the computation.

The experiments of this chapter show that there is only a small difference in performance between node marking and edge marking techniques, as implemented in Thor. Context-specific or type-specific swizzling appears unjustified.

Chapter 5

Prefetching

In the previous chapter, the client fetched only one object at a time from a server. This arrangement is simple, but the performance of single-object fetching is unacceptable [27, 33, 34]. To improve performance, the number of fetches must be reduced; some entity larger than a single object must be transferred from server to client. This chapter investigates techniques for the server to send more than one object at a time to the client. Although a more accurate name for these techniques would be "presending," I refer to these techniques by their more common name of *prefetching*.

Thor transfers groups of objects from server to client. On receiving a fetch request, a Thor server selects objects to send in response. The group of objects selected is called the *prefetch group*. Thor's dynamic selection of the group contrasts with most distributed object databases[4, 8, 9, 10], which cluster objects statically into pages and transfer pages in response to fetch requests. This chapter describes and compares different techniques for selecting prefetch groups.

The experiments presented in this chapter show that a good simple prefetcher is based on a breadth-first exploration of the object graph, with the exploration ending after the prefetch group reaches a designated number of objects. Performance of such a prefetcher can be improved by allowing the prefetcher to prune its exploration when encountering an object that has already been sent to the client. With the traversals measured, the best overall performance is attained when each prefetch group can contain up to 30 objects. With this level of prefetching, the dense traversal requires only about 33% of the time required for single-object fetching, and execution of the sparse traversal requires only about 51% of the time required for single-object fetching.

This dynamic prefetcher also does well when compared with a more static scheme that approximates page fetching. With a prefetch group size of 30, the prefetcher does about as well as a system using a static clustering of objects into 8-Kbyte pages; execution of the sparse traversal requires only about 17% of the time required for a system using that same static clustering.

Section 5.1 describes how a Thor server is structured, and how that structure affects prefetching.

Section 5.2 describes a number of different prefetching techniques and compares their performance using the current Thor implementation. Section 5.3 compares the performance of those prefetching techniques as the database size varies. The comparisons in Sections 5.2 and 5.3 use edge marking as the swizzling technique (see Chapter 4 for details). Section 5.4 revisits the issue of comparing node marking with edge marking in a system with prefetching. Section 5.5 reviews related work, and Section 5.6 summarizes the chapter.

5.1 Server and Prefetcher Structure

Each Thor server maintains information about which of its objects are present in each client cache. This information allows the server to invalidate objects in a client cache that have been made obsolete by updates at the server[46]. The same information is also used for efficient concurrency control [1] and distributed garbage collection [48]. The server's information always represents a superset of the objects actually in the client's cache: no object can be in the client's cache without being recorded at the server, but the server may record an object that is not in use at the client.

Each fetch request from the client causes the server to select a prefetch group containing the object requested and possibly some other objects. To simplify the structure of the server and the client/server interface, a fetch request is processed to completion, determining all members of the prefetch group, before any objects are sent to the client. The server does not split requests, prefetch "in the background," or otherwise perform computation except in response to a client request. When the sent objects arrive at the client, each must be entered into the Resident Object Table (ROT) and copied into the client cache.

Most of the prefetchers measured in this chapter are *dynamic* prefetchers: each dynamically computes a prefetch group at the time an object is requested. All of the dynamic techniques have access to the following information when the server receives the fetch request:

1. the identity of the object being requested;
2. which objects are currently in the server cache, and their size;
3. which of those objects are probably in the client cache; and
4. the maximum number of objects that the client will accept in the reply.

As noted in Chapter 1, all the experiments reported are set up in a way that ensures that all relevant server objects are in the server cache. For the experimental data presented in this and the remaining chapters, none of the costs (or variations in cost) are attributable to disk activity. Similarly, since the server and client are on the same machine, the network costs reflect only the cost of the protocols and scheduling, not transmission time. The data presented focus on the intrinsic computational cost of the implementation by (effectively) setting both the disk access time and network propagation time to zero. In spite of this simplification, the data is still dominated by the number of fetches.

In addition to the listed sources of information, a Thor server can always distinguish references from non-references in its stored objects. However, a Thor server does not usually associate any meaning with its stored objects, and it does not execute the method code that is associated with those objects. The object graph stored at a server is effectively a collection of variable-sized objects containing references to each other. However, all of the non-reference data is opaque to the server. This thesis uses the term *structural prefetching* to refer to prefetching that uses only the pointer structure of objects, with no other knowledge of the meaning of object fields.

5.2 Prefetching Techniques

Chapter 4 established that edge marking performs slightly better than node marking for cold traversals in the absence of prefetching. This section compares different prefetching techniques in some detail, assuming that edge marking is the swizzling technique being used. Section 5.4 returns to comparing node marking and edge marking in the presence of prefetching, to determine whether edge marking remains a better choice.

5.2.1 Hypotheses

Here are the three hypotheses about the performance of prefetching that are tested in this section:

1. *Simple structural prefetching significantly improves performance in Thor.*

   The intuition for this hypothesis is that it is expensive to fetch an object from the server, but the cost does not grow linearly with the number of objects. Bringing over a few "extra" objects that are "near" the fetched object does not increase the fetch cost much, but can eliminate fetches that would otherwise be needed. This hypothesis is supported by my experiments.

2. *Simple dynamic prefetching works as well as or better than fetching static page-like groups.*

   The experiments compare dynamic prefetching techniques to a static approximation of page fetching called *pseudopaging.* A static scheme must choose one clustering of objects into pages, whereas a dynamic scheme can choose prefetch groups based on the history of the computation.

   For example, consider the dense and sparse traversals described in Chapter 3. The sparse traversal is essentially a truncated form of the dense traversal. Assume that the clustering for the static scheme is based on the dense traversal: then one would expect a sparse traversal using the static scheme to fetch in extraneous objects. Intuitively, it would seem that a dynamic scheme would have more consistent performance across the dense and sparse traversals. At its best, the dynamic scheme might have performance comparable to the static scheme for the dense traversal. This hypothesis is supported by my experiments: simple breadth-first prefetching comes close to the performance of pseudopaging for a dense traversal, and exceeds the performance of pseudopaging for a sparse traversal.

3. *Information about object semantics improves prefetching performance in Thor.*
   Most of the prefetchers tested have no more information about an object than the number and location of its references to other objects. Adding hints about which fields to fetch should help performance when that information is accurate. This hypothesis is supported by my experiments, but the hints can easily degrade performance as well. Hints based on the dense traversal improve performance on the dense traversal to be near-optimal, but those hints have mixed results on the sparse traversal, compared to the best "uninformed" traversal.

5.2.2 Apparatus

The database and traversals used are identical to those used for the swizzling experiments of Chapter 4. In contrast to the single-object server used there, in this chapter the server has several different prefetch mechanisms. In the current implementation, the prefetch technique is selected at runtime by the client. The current implementation allows the prefetch technique and prefetch group size to vary on each fetch request sent from the client to the server. However, for all of the experiments reported in this chapter, both the prefetch technique and the prefetch group size are constant for a whole traversal.

Each new (non-duplicate) object fetched to the client must be both copied into the client cache and entered into the resident object table, whether it is subsequently used by the computation or not. In these experiments, client cache management never takes place; the client cache is large enough so that no storage management is required. Chapter 7 considers the problem of combining prefetching with storage management. A few configurations required a larger cache than was possible with the platform used; I report only the data that could be collected from the system that I had.

As in Chapter 4, the client and server are running as separate processes on a single DEC 3000/400 workstation. The client application is in the same process as the client. The times reported are elapsed (wall-clock) times, as measured by the on-chip cycle counter of the workstation.

The rest of this section describes the seven prefetching techniques implemented and measured. Those techniques are:

- perfect
- pseudopaging
- bf-cutoff

- bf-continue
- bf-blind
- depth-first
- class-hints

Perfect prefetching uses knowledge of the future computation (a recorded trace) to send the objects that will be used next. Perfect prefetching is not a practical technique, but it serves as a useful comparison for techniques that can be implemented.

Pseudopaging does not dynamically compute a prefetch group at run time. Instead, at database creation time, a clustering process computes fixed size prefetch groups, bounded by a fixed number of bytes, at database creation time.

Objects are clustered according to their creation order. All CompositeParts are created before any Assemblies, and each CompositePart's contained AtomicParts and Connections are created as part of the processs of creating CompositeParts. Thus AtomicParts and Connections tend to be clustered together with their parent CompositePart, CompositeParts tend to be clustered with other CompositeParts, and Assemblies tend to be clustered with other Assemblies. Clustering in this creation order is a reasonable clustering for the dense traversal, as well as being simple to implement. Pseudopaging is intended to be an approximation of the prefetching performance of a realistic page-fetching system; accordingly, the clustering is not optimal, but it is not particularly bad either. In particular, this creation-order clustering is better than two of the static clusterings (breadth-first and depth-first) used by Stamos [67] in his study. It is less effective than what could be achieved by the most sophisticated clustering techniques [71], but it is also much less demanding of both computation and insight into the database. The average pseudopage has about 60 objects in it, so we would expect dynamic prefetching of groups of 60 or more objects to beat pseudopaging.

There are three variants of breadth-first prefetching implemented; each has a name starting with "bf". *Bf-cutoff prefetching* dynamically computes a prefetch group by exploring the object graph breadth-first from the object being fetched. The prefetcher prunes the exploration on encountering any object already present at the client; that is, it does not explore any of the references contained in that object.

Bf-continue prefetching dynamically computes a prefetch group by exploring the object graph breadth-first from the object being fetched. Like bf-cutoff, bf-continue does not send any object already present at the client. Unlike bf-cutoff, bf-continue does continue to explore the references contained in such an object.

Bf-blind prefetching is yet another variant of breadth-first prefetching. Unlike bf-cutoff, it does not prune its exploration. It pays no attention to whether an object is likely to be present at the

Class	Field(s) prefetched	Class prefetched	Refs prefetched	Total refs
AtomicPart	outgoing	vec[Connection]	1	4
BaseAssembly	privChildren	vec[CompositePart]	1	5
ComplexAssembly	children	vec[Assembly]	1	4
CompositePart	root	AtomicPart	1	6
Connection	to	AtomicPart	1	2
vec[Assembly]	*first 3 elements*	Assembly	3	3
vec[CompositePart]	*first 3 elements*	CompositePart	3	3
vec[Connection]	*first 3 elements*	Connection	3	3

Figure 5-1: Details of the class-hints prefetcher

client when doing the prefetch. Accordingly, it has a lower overhead at the server, but is likely to send redundant objects to the client.

Bf-cutoff and bf-blind represent opposite extremes in terms of filtering out redundant objects. Bf-cutoff filters at the server and prunes exploration quickly, while bf-blind filters at the client and doesn't prune. Bf-continue represents a middle position, filtering at the server (like bf-cutoff) but not pruning (like bf-blind).

Depth-first prefetching dynamically computes a prefetch group by traversing the object graph depth-first from the object being fetched. As with bf-cutoff, the depth-first prefetcher prunes the traversal on encountering any object that is already present at the client.

Class-hints prefetching uses information about the class of each object to control the prefetching process. The class of each object is readily determined at the server. The server has a small table mapping each class to information about which fields should be prefetched, presumably because those are the fields most likely to be used in the near future by the client application. Only the children contained in those fields are considered by the prefetcher, which then repeats the same process for each child, based on that child's class. In the implementation for which data is presented, the class-hints table is based on the dense traversal. A summary of the fields of database objects appears in Chapter 3. A summary of the hints used appears in Figure 5-1, which also indicates the number of references that are indicated as "good to fetch" and the total number of references that could be fetched. The prefetcher is otherwise built (logically) on the bf-cutoff prefetcher; the class-hints prefetcher can be seen as a filter on bf-cutoff that causes some irrelevant children to be omitted.

5.2.3 Results

I begin by comparing perfect prefetching and pseudopaging. Figure 5-2 shows the comparison between these techniques for both dense and sparse traversals, as a function of the prefetch group

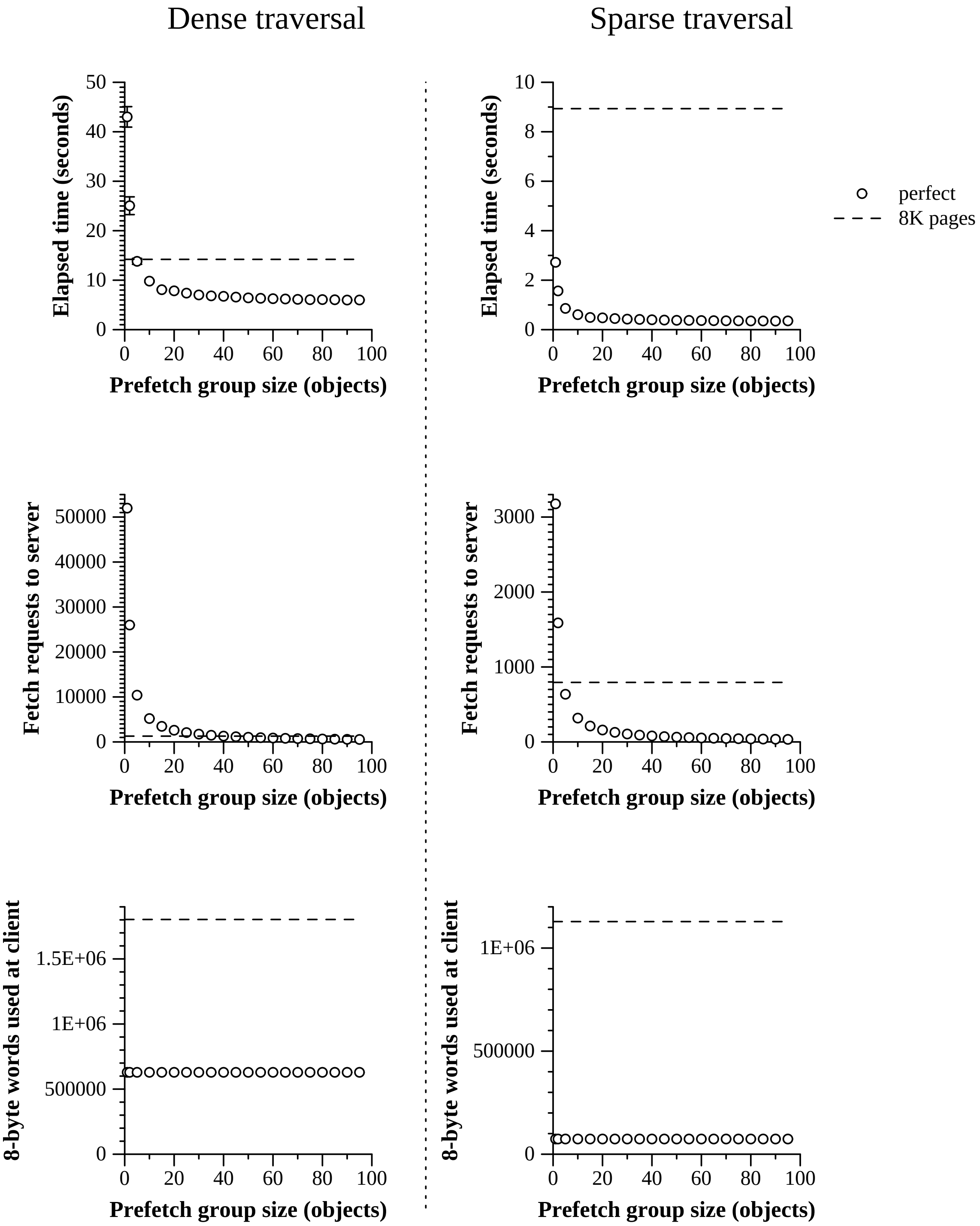

Figure 5-2: Perfect vs. pseudopaging

size. Recall that the prefetch group size is the maximum number of objects that may be sent to the client in response to each fetch request.

There are two columns of graphs: the left column shows the data for the dense traversal, the right column shows the data for the sparse traversal. In both cases the top graph represents elapsed time for the traversal, the middle graph represents the number of fetch requests sent to the server, and the bottom graph represents the total space occupied in the client object cache at the end of the traversal. For the graph of elapsed time, I have plotted error bars representing a 95% confidence interval (that is, 2 sample standard deviations on either side of the mean). If the error bars appear to be absent, it is because the error bars are smaller than the mark used for the mean. Generally the variances are quite small. Neither the number of fetch requests nor the storage used vary across trials of a single configuration, so there are no error bars plotted for those graphs.

Since pseudopaging sends constant-sized object groups, it is not affected by the prefetch group size, and appears on the graphs as a horizontal line. Pseudopaging does well on the dense traversal and poorly on the sparse traversal, which is as expected since the clustering used for pseudopaging was intended to support efficient dense traversals. Although the clustering brings over about three times as many bytes as are needed with perfect knowledge of the computation, it brings over less than one third of the database for the dense traversal and less than 20% of the database for the sparse traversal. On the dense traversal, perfect prefetching performs better than pseudopaging as soon as 5 or more objects can be returned for each client request. Perfect prefetching quickly flattens out with increasing prefetch group size: there is very little improvement in performance for group sizes larger than 15-20 objects. One may conclude that for the implementation and workload studied, a relatively small prefetch group size suffices if one can pick those objects perfectly.

Examining perfect prefetching and pseudopaging establishes a basis for considering dynamic prefetchers. The perfect prefetcher is as good as any dynamic prefetcher can be, while pseudopaging gives a sense of what can be achieved with a simple static technique. There is no point in looking for prefetchers that do better than the perfect prefetcher, and there is no point in considering elaborate dynamic prefetchers unless they can do significantly better than the simple paged system. The goal is to find simple dynamic techniques that can fill the gap between pseudopaging and perfect prefetching.

Figure 5-3 compares bf-cutoff with the perfect prefetcher and pseudopaging. Bf-cutoff gets good performance for small sizes of prefetch group, but its performance degrades for large group sizes, in a rather surprising way. As can be seen from the number of fetch requests, the dense traversal with bf-cutoff actually requires more fetches for a prefetch group of 60 objects (17142 fetches) than for a prefetch group of 10 objects (13123 fetches). The explanation is found in the relatively small size of the database. Prefetch group sizes larger than 30 and less than 80 tend to prefetch all of the internal Assemblies of the Assembly hierarchy, leaving only isolated leaves to be fetched one by one

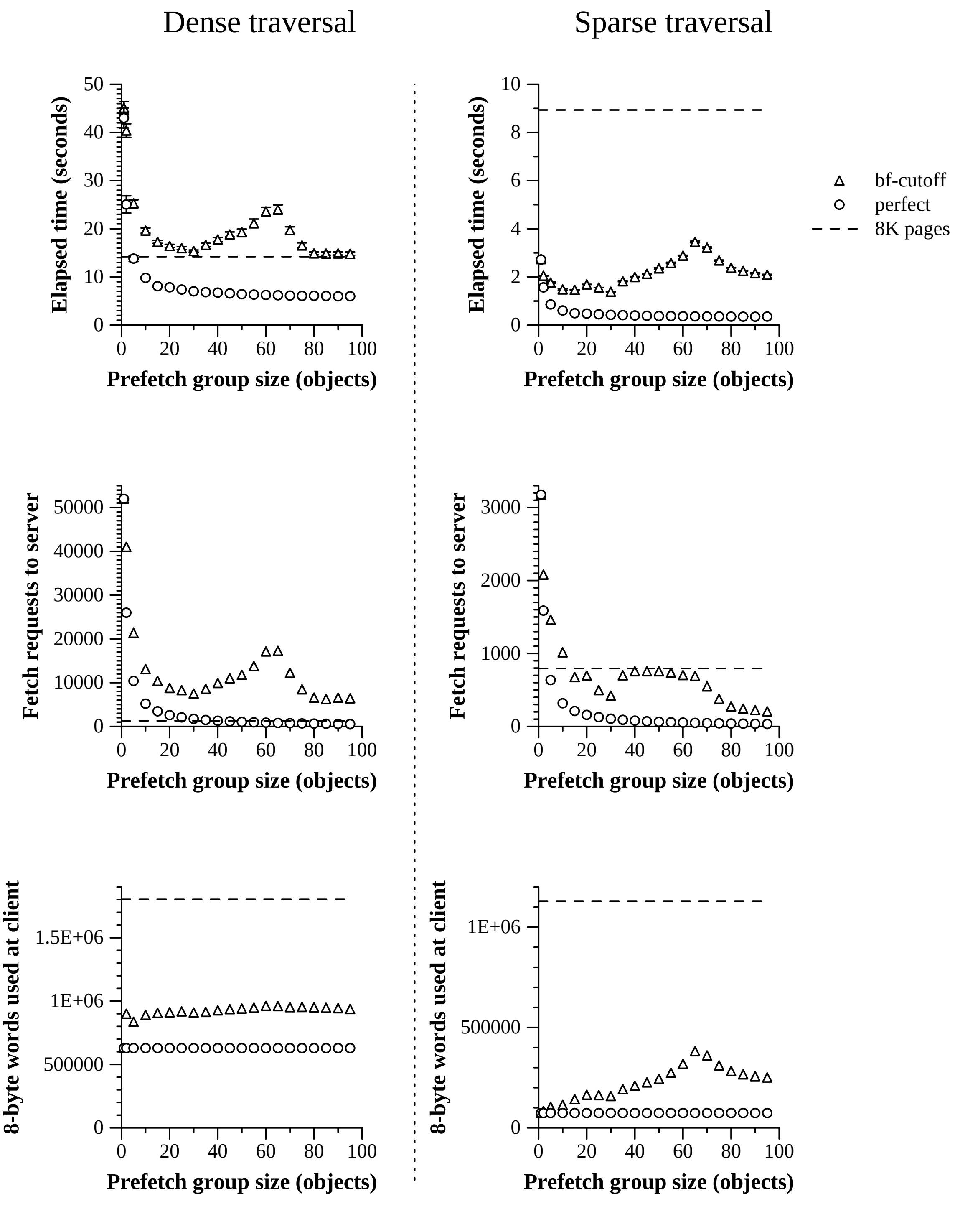

Figure 5-3: Bf-cutoff vs. perfect and pseudopaging

in the later parts of the traversal. Accordingly, I call this performance degradation *leaf-fetching*.

The sparse traversal shows a similar degradation, but the underlying cause is different. Leaf-fetching is characterized by a sharp increase in fetches. Instead of that sharp increase, there is a plateau in the graph of fetches (middle right). The spike in the elapsed time is not caused by a spike in the number of fetches. Instead, as shown by the space graph for the sparse traversal (lower right), there is a sharp increase in the number of bytes used. The performance of the sparse traversal degrades simply because there are more irrelevant objects being sent.

Figure 5-4 shows the performance of bf-continue. The bf-continue prefetcher eliminates the peak in fetches that occurs with bf-cutoff. Continuing to explore children even when the parent is already at the client increases the likelihood that reachable objects will be prefetched, rather than being left as isolated leaves. If we were to ignore elapsed time and consider only the number of fetches, we would conclude (incorrectly) that bf-continue does very well on the dense traversal, and reasonably well on the sparse traversal. However, each fetch itself has become significantly more expensive, since the server can continue for long periods of time trying to find an object that is not already at the client. The expense is not in the table lookup, but in the repeated discovery and exploration of objects that have already been sent to the client. As a result, the elapsed time of bf-continue is worse than bf-cutoff for the dense traversal, except at the nadir of bf-cutoff's bad performance. In terms of both elapsed time and space used, bf-continue performs worse on the sparse traversal than bf-cutoff. As with bf-cutoff, the problem is that irrelevant objects are being fetched; however, bf-continue brings over even more of these irrelevant objects than bf-cutoff does.

Figure 5-5 shows the performance of bf-blind prefetching. As with bf-continue, bf-blind prefetching eliminates leaf-fetching, and shows better performance than bf-continue for prefetch group sizes of 50 objects or fewer. However, it consistently performs worse than bf-cutoff in terms of elapsed time. Interestingly, its usage of space with increasing prefetch group size is better for the sparse traversal than that of bf-cutoff. This effect occurs in a situation where bf-cutoff prunes its exploration and prefetches some irrelevant objects, whereas the blind breadth-first refetches a group of largely redundant objects. Redundant objects impose a penalty in terms of transmission costs and processing time, but they do not occupy space in the cache or resident object table. In contrast, useless prefetched objects have both kinds of costs.

Figure 5-6 shows the performance of depth-first prefetching. As noted earlier, this depth-first prefetcher uses a model of the client cache to prune its exploration of the server's objects, much as bf-cutoff does. The performance of the depth-first prefetcher improves quickly with increasing group size, up to a group size of about 10. With larger group sizes, depth-first prefetching shows little improvement on the dense traversal and a marked degradation on the sparse traversal, as more and more irrelevant objects are prefetched into the client cache.

The poor performance of a depth-first prefetcher may seem surprising, since the traversals being

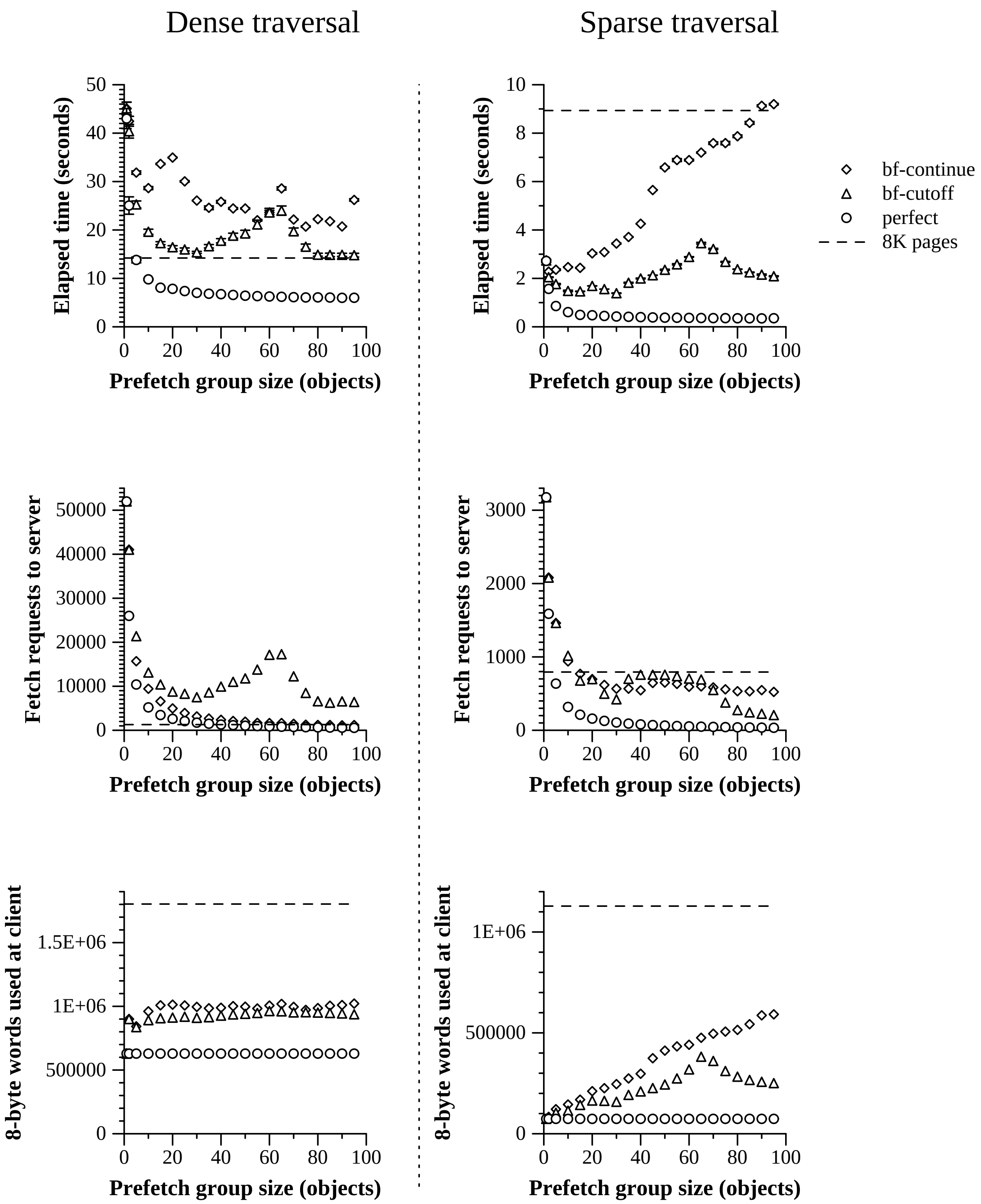

Figure 5-4: Bf-continue vs. bf-cutoff

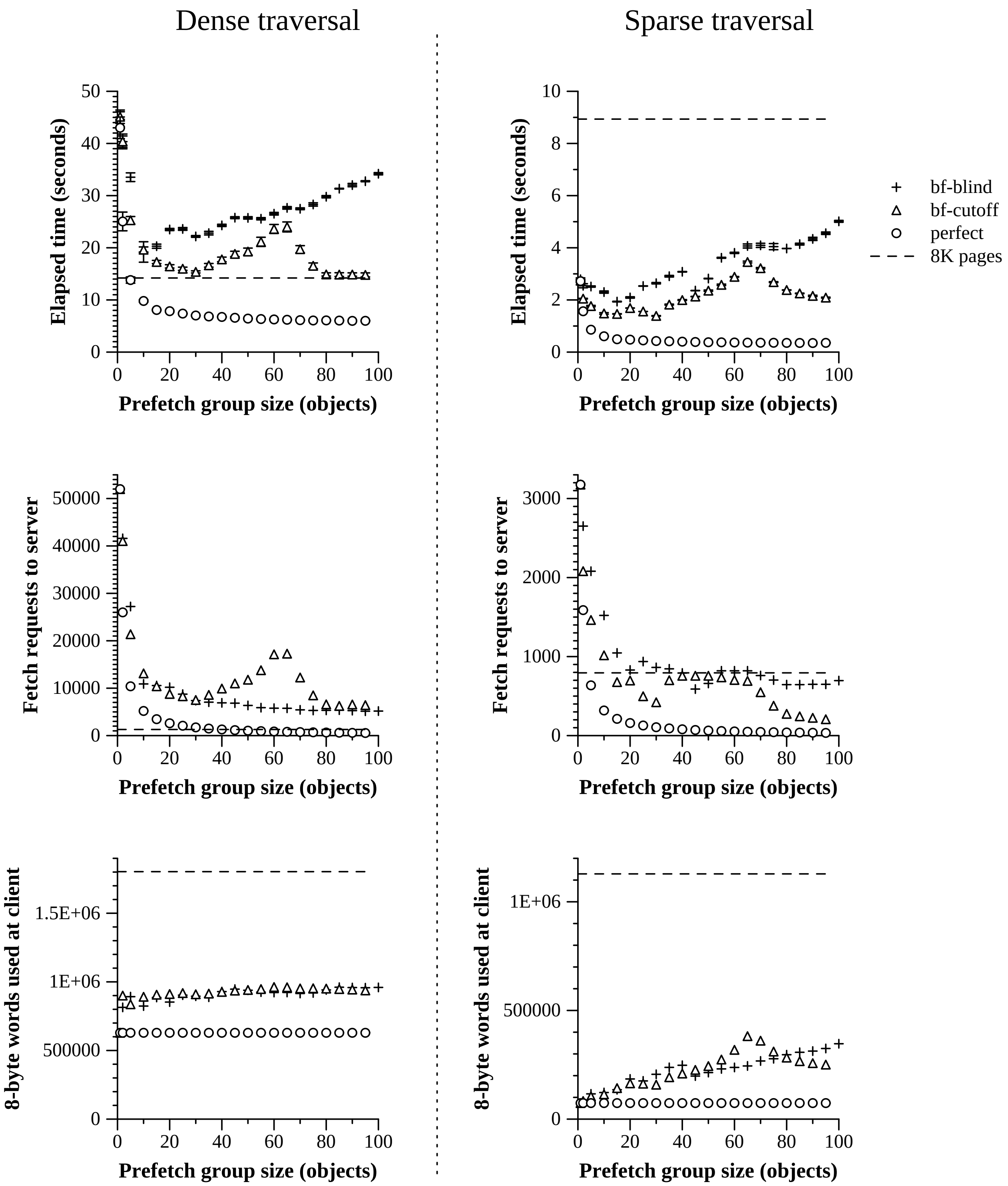

Figure 5-5: bf-blind vs. bf-cutoff

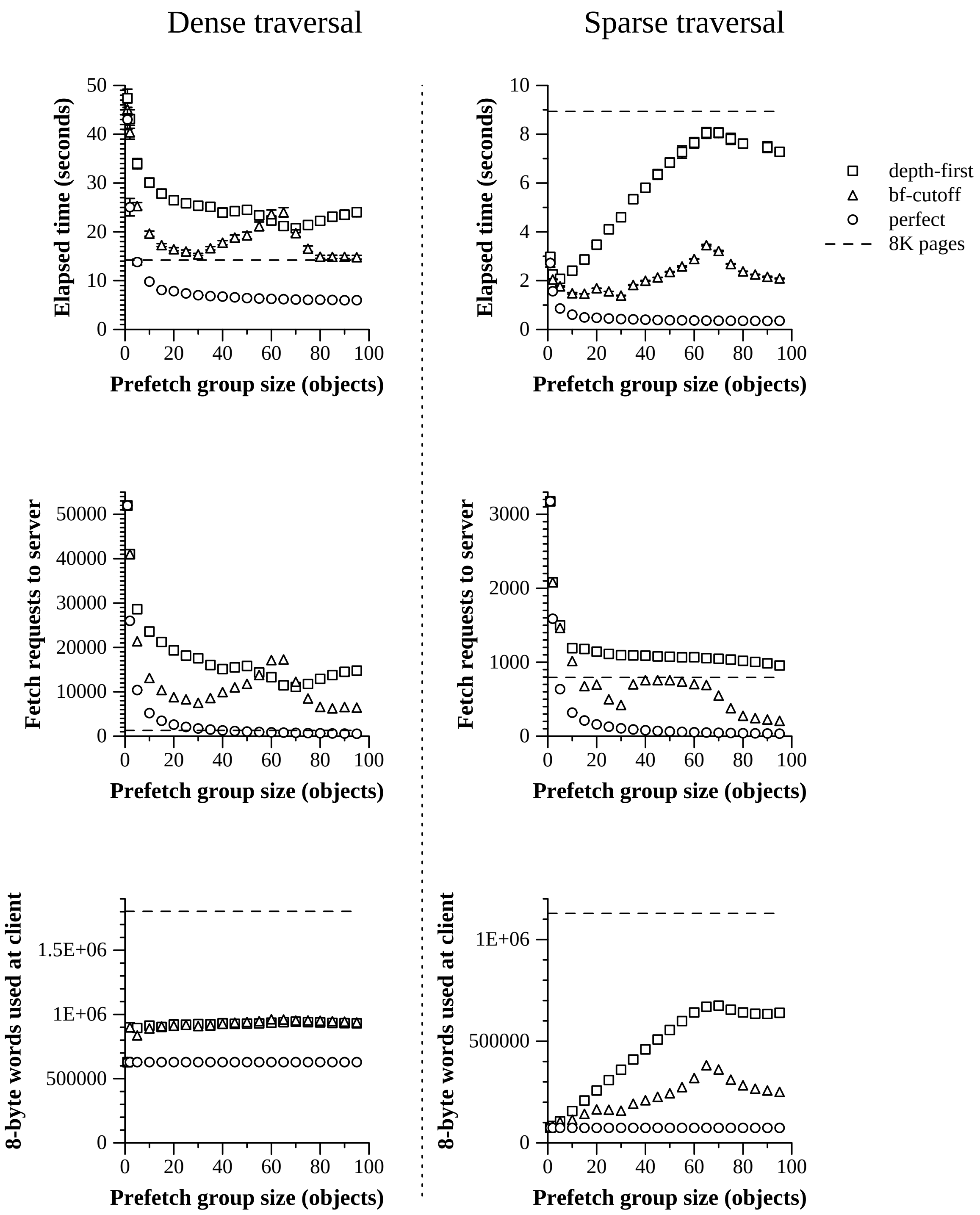

Figure 5-6: Prefetching: depth-first vs. bf-cutoff

executed at the client are described as "depth-first". But although the traversal performed by the client is logically depth-first, it is not depth-first in terms of the representations of objects as stored at the server (a distinction previously made in Chapter 3). Each object contains numerous pointers to parent objects and containing objects, and those pointers are not touched by the traversals being measured. For example, only one field of an AtomicPart is used in the traversals: a pointer to a vector of outgoing Connections. However, each AtomicPart also has a pointer to a vector of incoming Connections, a pointer to its parent object, and a pointer to a string (containing its "type" with respect to the OO7 database). A structurally depth-first traversal of these various pointers does not bear much resemblance to the logically depth-first traversal that the client is performing.

The online dynamic techniques presented so far — bf-cutoff, bf-continue, bf-blind, and depth-first — have used only the structure of the representation at the server. It is interesting to consider what sort of benefit can be gained from having more information available at the server. Figure 5-7 compares class-hints prefetching with optimal prefetching and pseudopaging. Class-hints does well for the dense traversal: its fetch requests and storage used are nearly optimal, and its elapsed time is very good. However, it performs poorly on the sparse traversal: the space graph shows that it fetches too many objects, and the fetches graph shows that even with all those objects being fetched, the right objects are not being prefetched for many group sizes.

Figure 5-8 directly compares class-hints prefetching with the underlying bf-cutoff prefetcher. The effect of the additional information is clearly beneficial for the dense traversal. However, on the sparse traversal, the fetches graph shows that the hints don't make much difference. The hints cause too many objects to be fetched, until the prefetch group size is about 80. When the prefetch group is that size or larger, each chunk of objects fetched is large enough that it contains most of the objects that are needed in the near term.

The graphs in Figure 5-8 show that the number of client requests to the server is similar for class-hints and bf-cutoff prefetching. However, the server responses are quite different: both the storage used at the client and the elapsed time differ between the two prefetchers in otherwise-identical configurations.

5.3 Larger Databases

The previous experiments have shown that the bf-cutoff prefetcher works well for these traversals, especially when using a maximum prefetch group size of about 30 objects. This section describes simple experiments intended to determine how the best prefetch group size changes (if it changes) with larger databases. All the previous experiments have shown results for a database with 20 AtomicParts per CompositePart, and 3 Connections per AtomicPart. This section considers two different ways of making the database larger: first, by increasing the number of Connections per

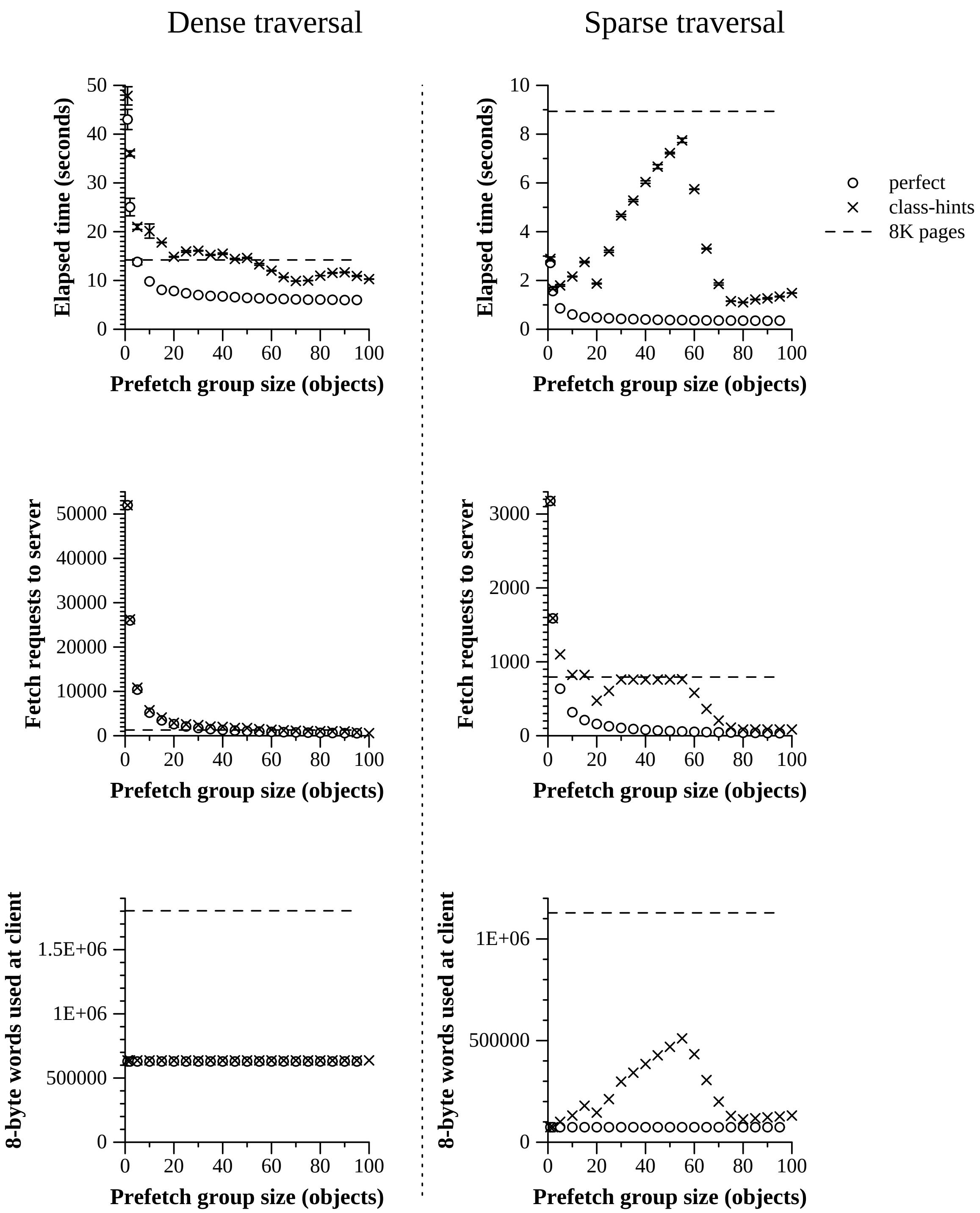

Figure 5-7: class-hints vs. perfect and pseudopaging

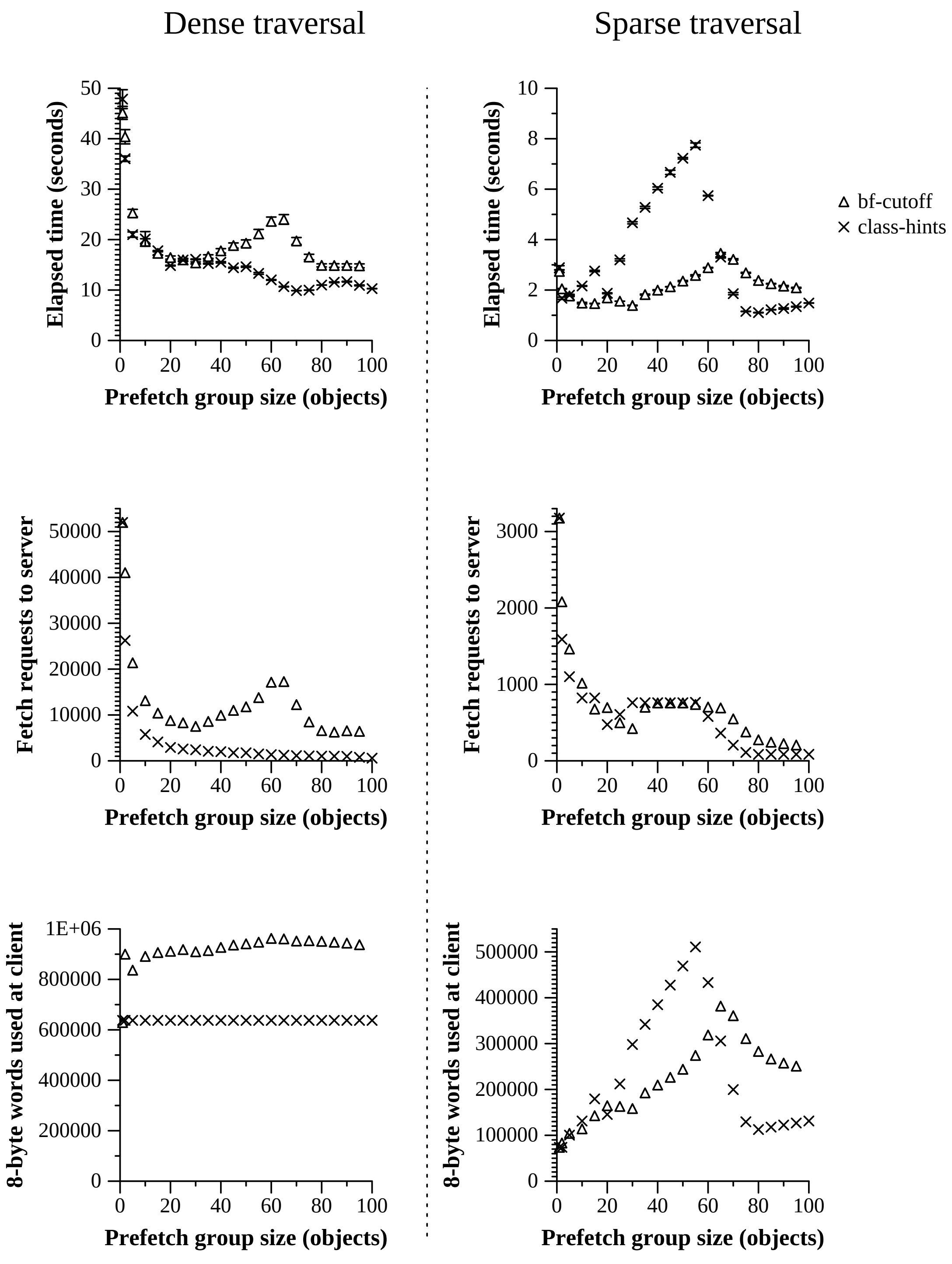

Figure 5-8: class-hints vs. bf-cutoff

AtomicPart, and second, by increasing the number of AtomicParts per CompositePart.

5.3.1 Hypotheses

The following three hypotheses are tested in this section:

1. *The best prefetch group size is independent of the size of the database.*
   One would not expect the number of objects at the server to affect the granularity of fetching that is most efficient for the particular object structure and workload. This hypothesis is supported by my experiments.

2. *The best prefetch group size is dependent on the size and structure of objects in the database.*
   One would expect differently-arranged configurations of objects to behave differently, and for the best prefetching approach to be different. This hypothesis is supported by my experiments.

3. *Bf-cutoff is the best dynamic prefetch technique that does not depend on knowledge of object usage, regardless of changes in database size or object structure.*
   The intuitive explanation of bf-cutoff's advantage is that it represents the best approximation of the "next" objects to be referenced. This explanation suggests that it should do better than the other techniques, with the possible exception of class-hints (which uses additional information). Further, bf-cutoff should do better regardless of changes in the size or structure of the database. This hypothesis is supported by my experiments.

5.3.2 Apparatus

The first set of experiments uses a database (called DoubleConnections) that is identical to the previously-used database, but with double the number of Connections per AtomicPart (6 instead of 3). The second set of experiments uses databases (called DoubleParts and TripleParts) that are identical to the previously-used database, but with double or triple the number of AtomicParts per CompositePart, (40 or 60 instead of 20).

How do these changes affect object size and database size? The following are some approximate calculations based on the fact that AtomicParts and Connections far outnumber CompositeParts in the database. In the DoubleConnections database, the number of objects fetched for a dense traversal is increased by a factor of roughly 1.75, and the size of the average object is increased by a factor of roughly 1.25. In the DoubleParts (TripleParts) database, the number of AtomicParts per CompositePart nearly doubles (triples) the number of objects fetched for a dense traversal, while hardly affecting the size of the average object.

5.3.3 Results

Figure 5-9 shows optimal and breadth-first prefetching for the DoubleConnections database. Figure 5-10 compares the variants of breadth-first prefetching, and Figure 5-11 compares breadth-first to the other realizable policies. For the dense traversals, the best prefetcher is still breadth-first; however, the best elapsed time is now at a prefetch group size of about 10. The class-hints prefetcher does very poorly. It is set up for a database that has 3 Connections per AtomicPart, and prefetches 3 Connections with each AtomicPart; so it is prefetching only half as many Connections as it should. It is easy to fix this problem and have good class-hints performance again, but the degradation shows that the class-hints prefetcher is quite brittle when compared to techniques that do not expect to have any knowledge of the "right" things to fetch.

The sparse traversals are much the same as before, which is as expected since the sparse traversal is not affected by the number of Connections per AtomicPart.

For the remaining experiments of this section, I present only the elapsed time of the various prefetchers, and only on the dense traversal. Figure 5-12 compares elapsed times for various prefetching techniques on a database with 20 AtomicParts per CompositePart (that is, the "normal" database). There are two graphs in Figure 5-12, but both have the same axes. Rather than putting all five curves on a single graph, three of the five prefetching techniques are presented on each graph for clarity (breadth-first is repeated on both top and bottom graph to allow comparison).

Figure 5-13 compares elapsed times for the prefetching techniques on a database with 40 AtomicParts per CompositePart and Figure 5-14 shows the same comparison for a database of 60 AtomicParts per CompositePart. The number of Connections per AtomicPart is still 3 in these databases, so they have 60, 120 or 180 Connections per CompositePart.

For the dense traversal, as the number of AtomicParts per CompositePart grows, the breadth-first prefetcher has the best performance overall. Class-hints does better for the smallest database (the one for which it was designed), but loses its advantage as the underlying database diverges further from the structure to which it is tuned. Around a prefetch group size of 60 objects, there is an increase in elapsed time for the breadth-first traversal, due to leaf-fetching, apparent in Figure 5-12. That increase does not appear in Figures 5-13 and 5-14, suggesting that the leaf-fetching phenomenon at that object group size is a quirk of a particular database size. Both bf-continue and depth-first perform worse relative to the performance of breadth-first with increasing numbers of AtomicParts per CompositePart. A prefetch group size of 30 still seems reasonable for good performance with the breadth-first prefetcher, even as the number of AtomicParts per CompositePart increases.

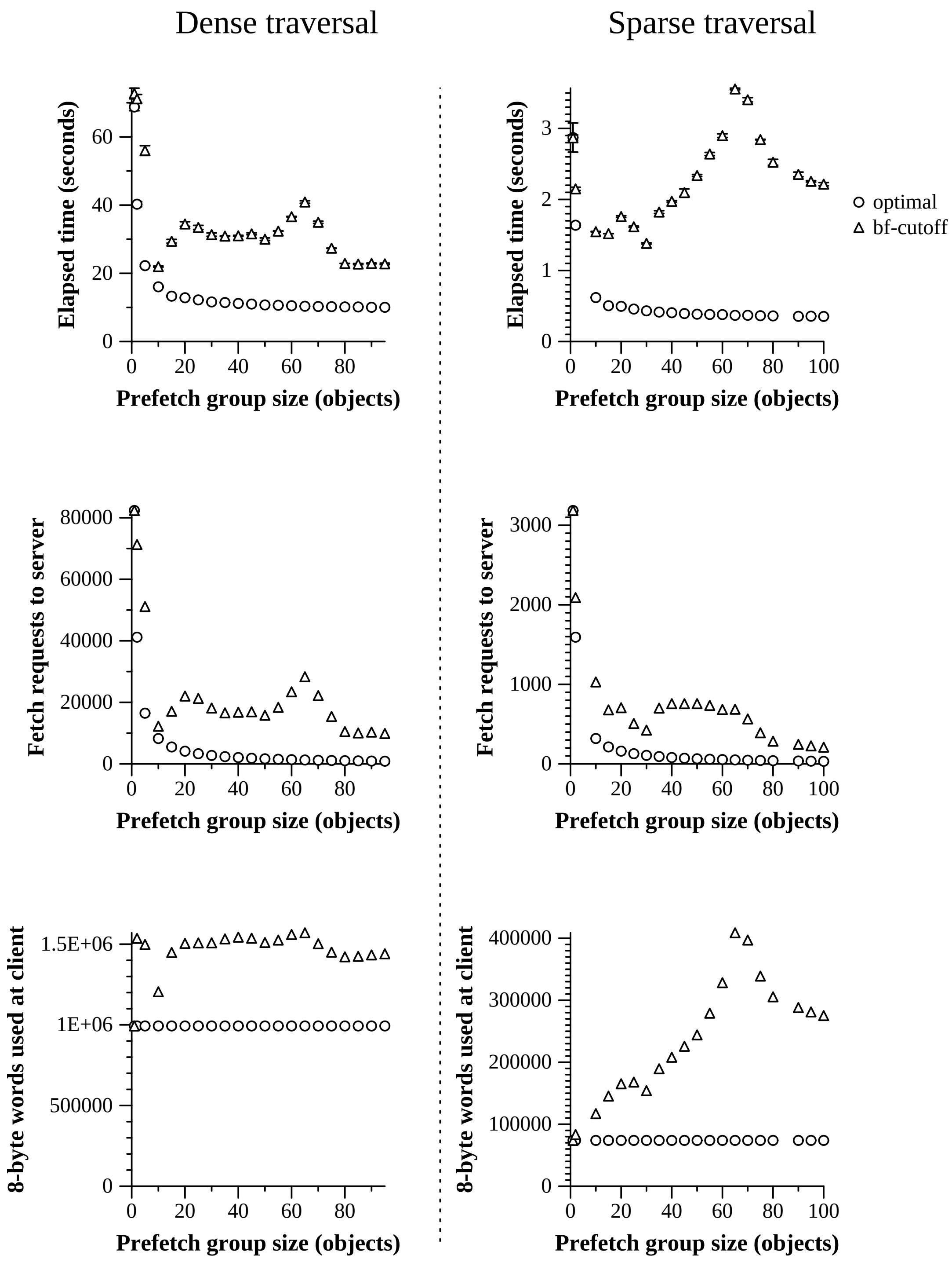

Figure 5-9: Optimal vs. breadth-first (DoubleConnections)

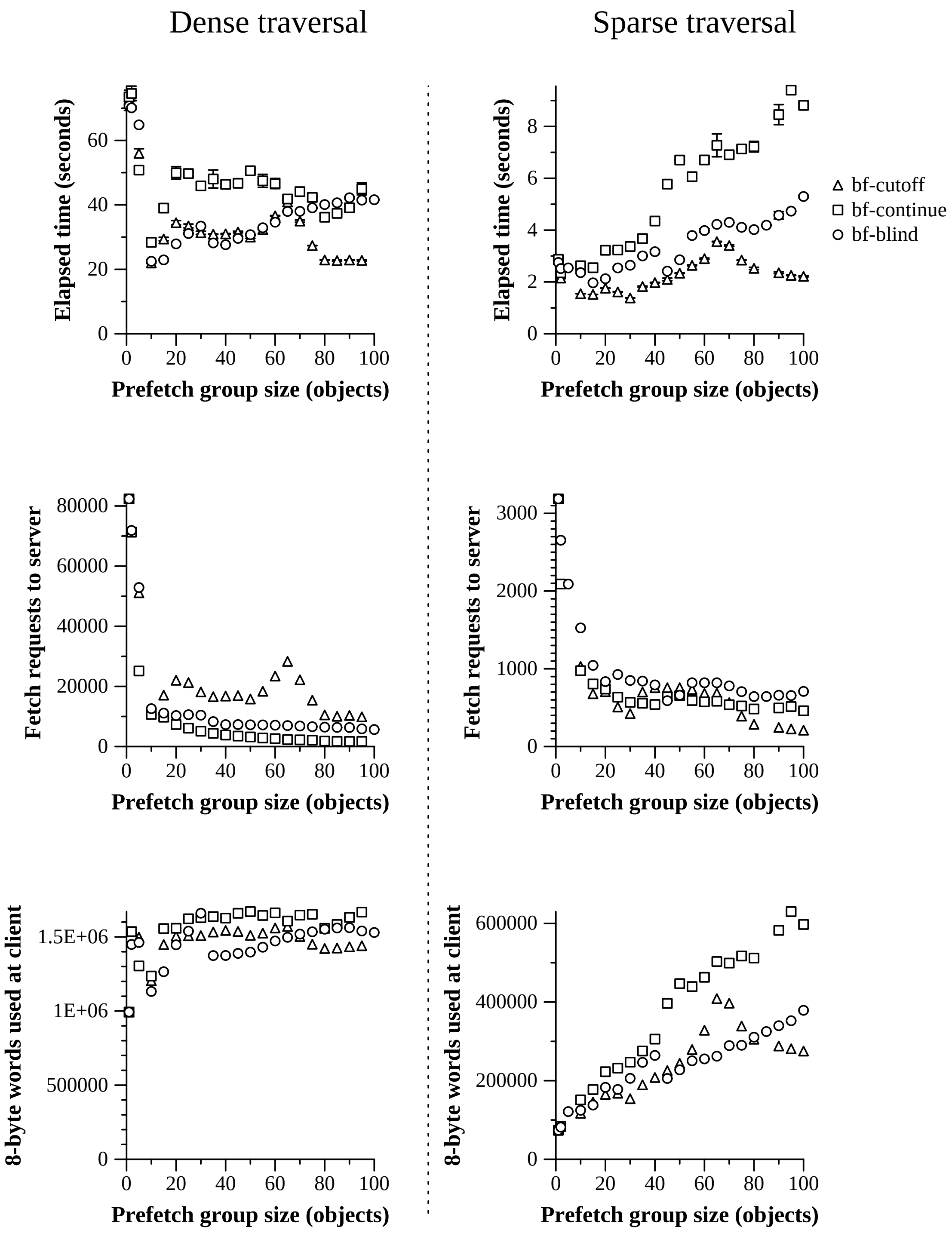

Figure 5-10: Breadth-first variants (DoubleConnections)

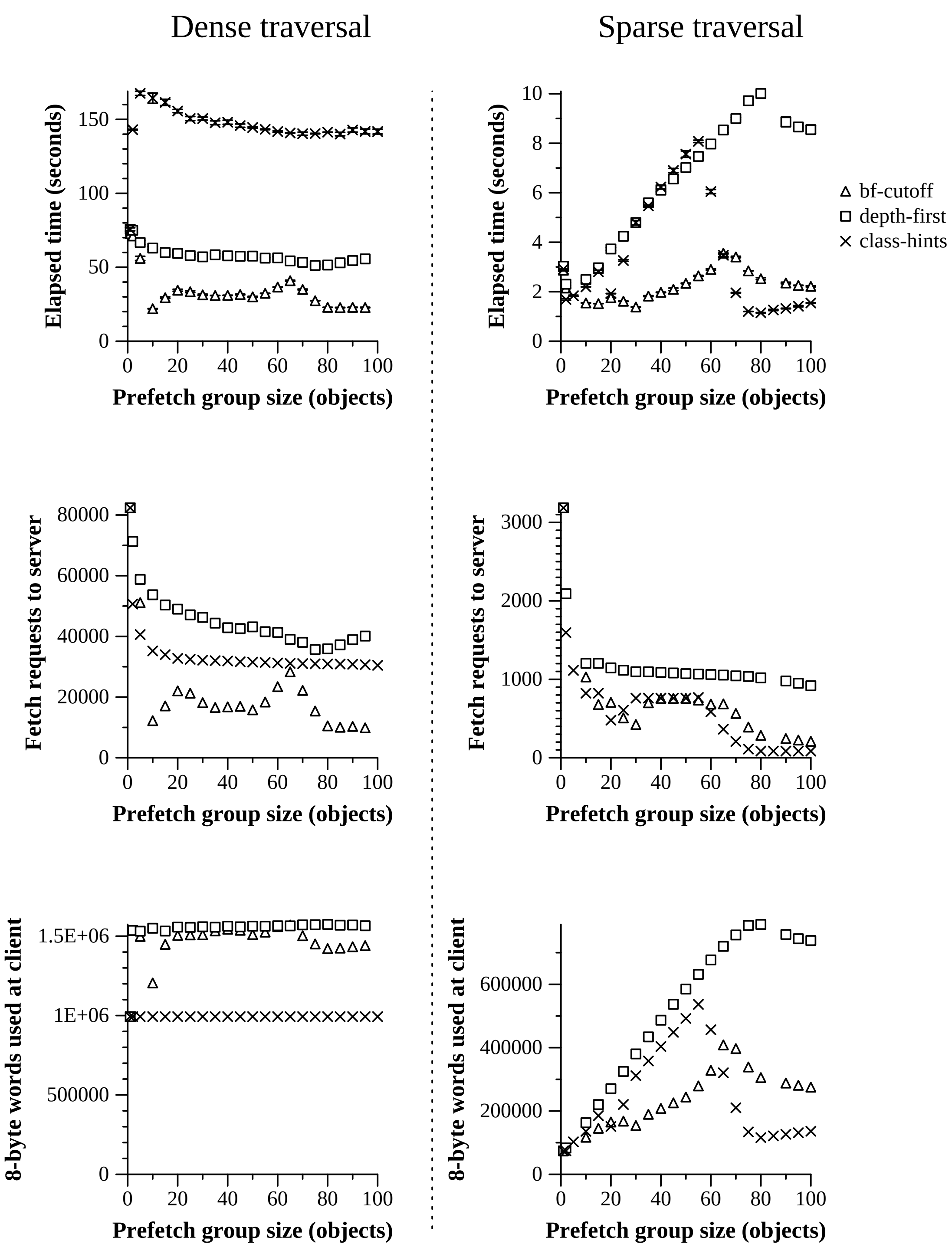

Figure 5-11: Other prefetchers (DoubleConnections)

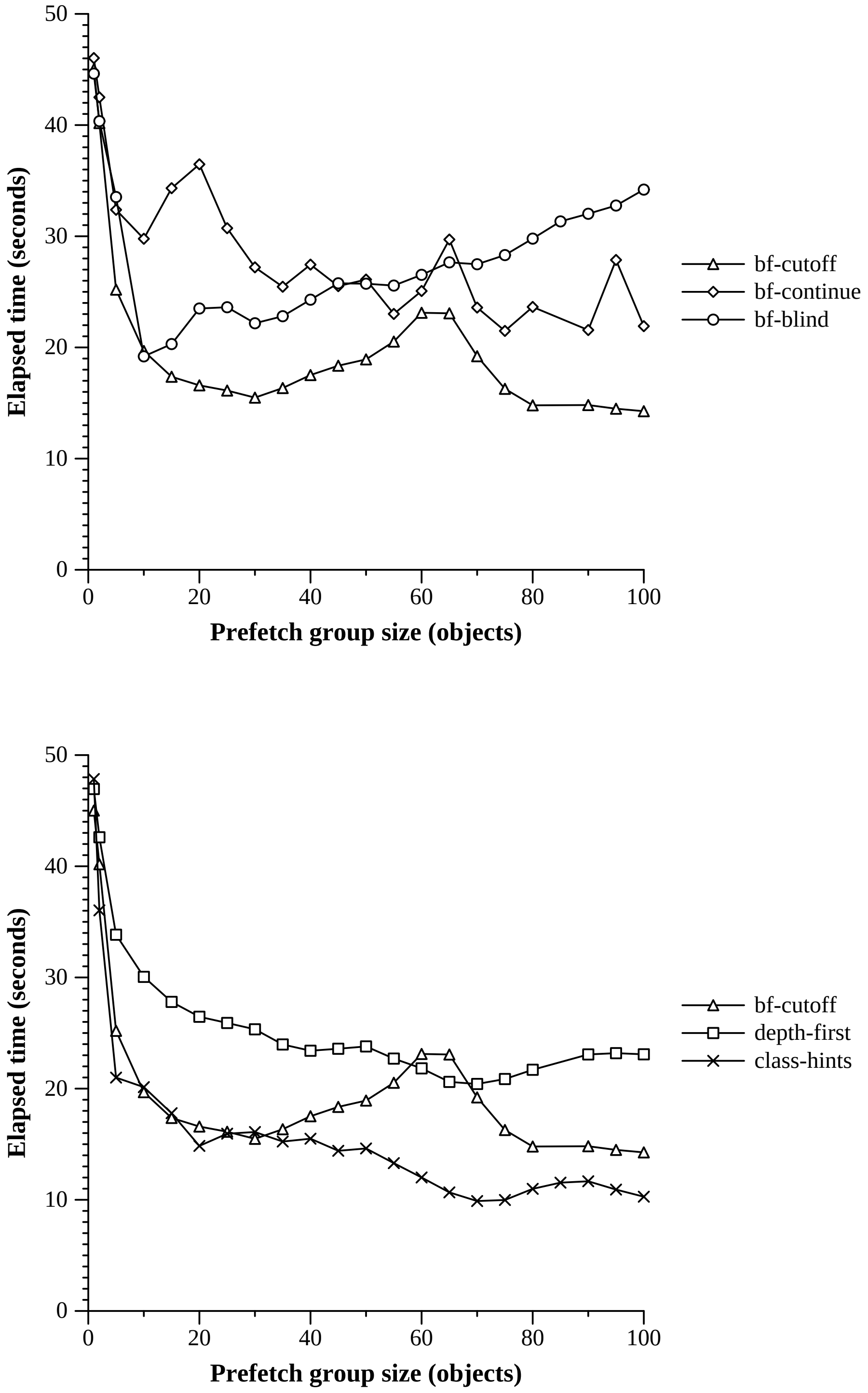

Figure 5-12: Elapsed time vs. prefetch group size, normal database

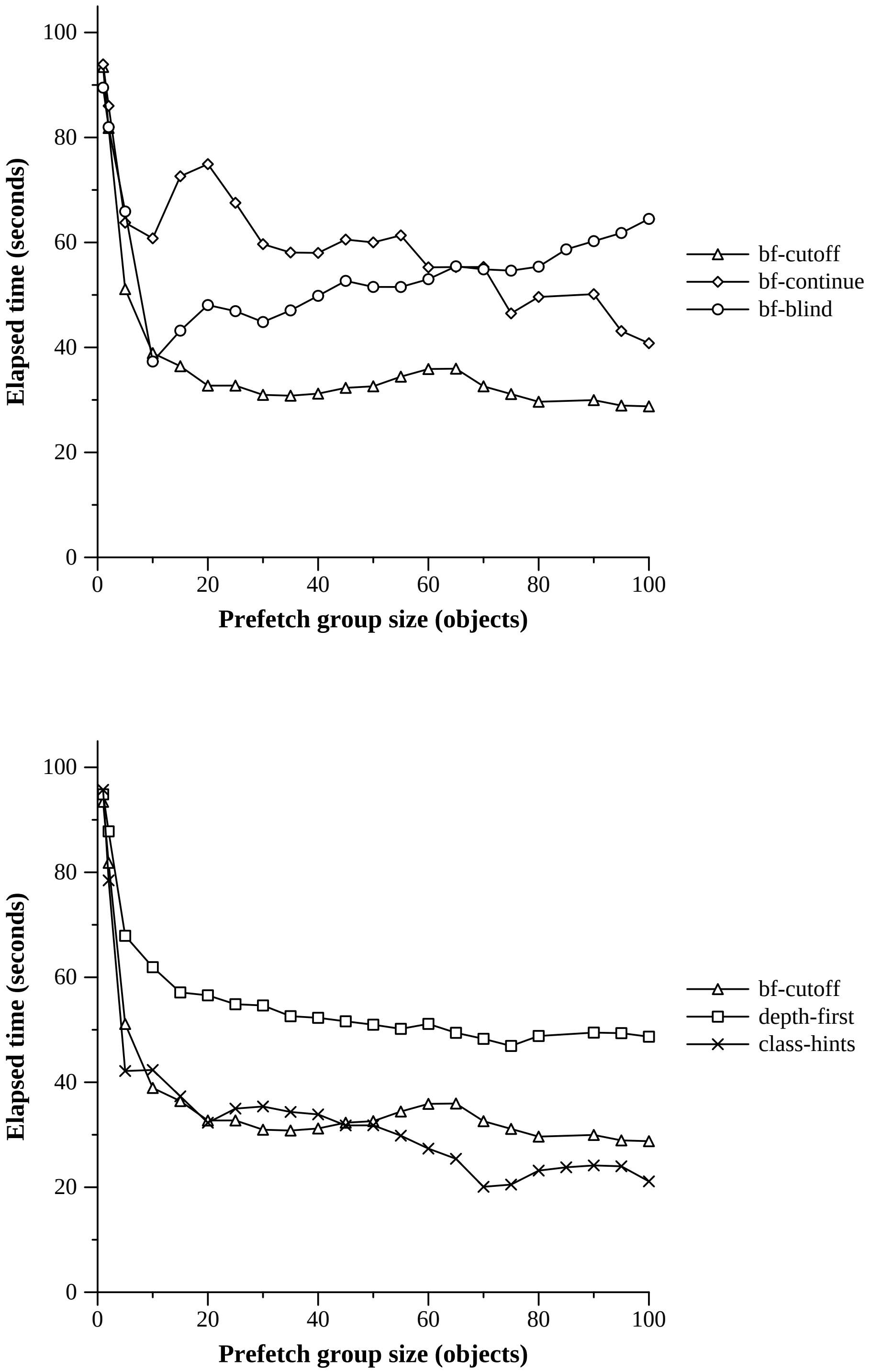

Figure 5-13: Elapsed time vs. prefetch group size, DoubleParts

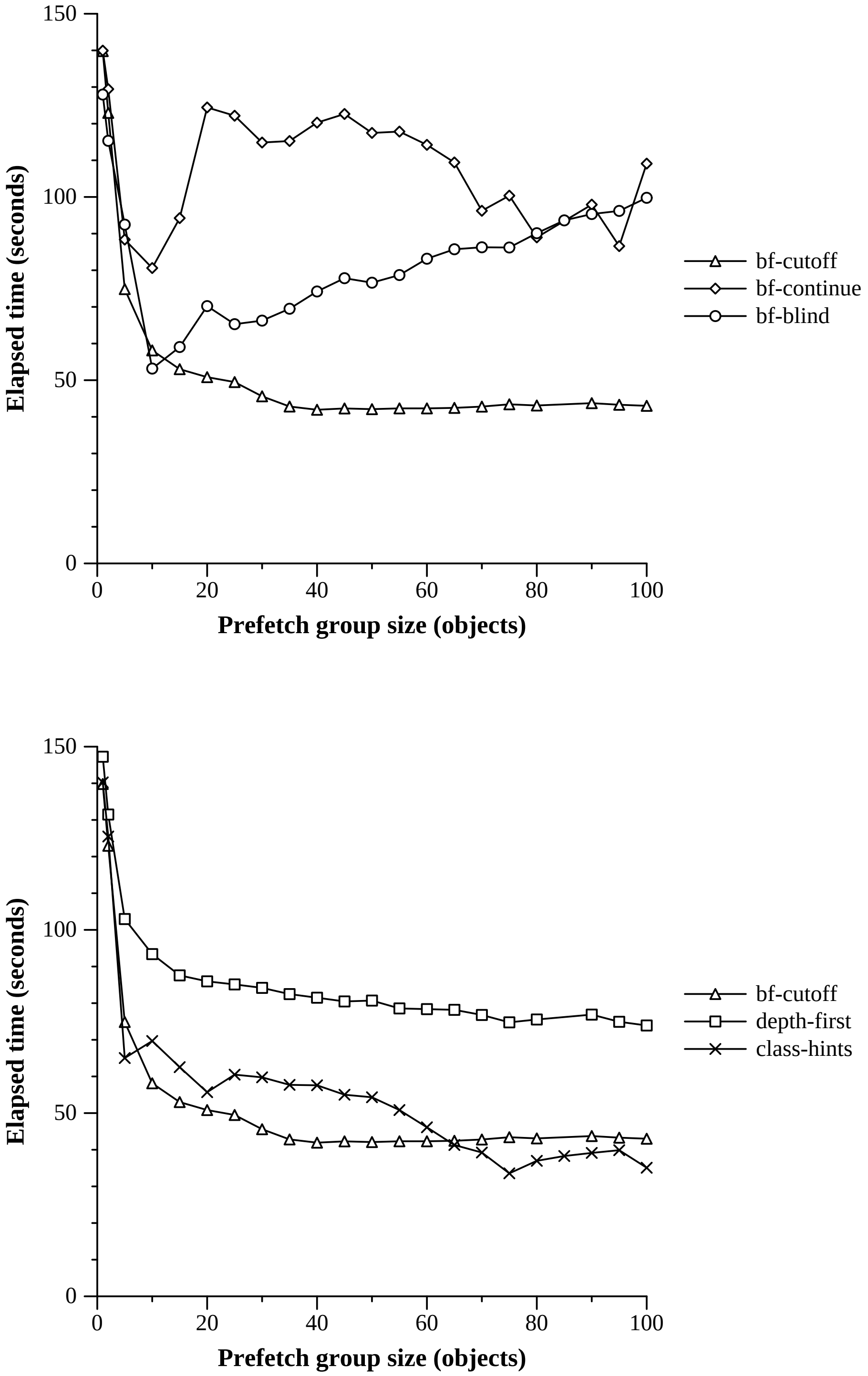

Figure 5-14: Elapsed time vs. prefetch group size, TripleParts

5.3.4 Summary

For the original database used, a breadth-first prefetcher with a prefetch group size of 30 gave the best overall performance. With the DoubleConnections database, the best prefetch group size for the breadth-first prefetcher drops to 10; but for the DoubleParts database, the best prefetch group size is approximately unaffected.

There is no simple explanation apparent for the relationship between a good prefetch group size and object size. If the determining factor were the size of packets transmitted from server to client, for example, one would expect the inverse relationship between object size and prefetch group size to be more linear. When the average object size increases by 25%, we would expect the prefetch group size to drop to 0.8 of what it was, to a prefetch group size of 24. Instead, the "good" prefetch group size drops to 10.

Since both DoubleConnections and DoubleParts have more objects than the original database, but only DoubleConnections significantly changes the average object size, it seems reasonable to argue that average object size is more important than the number of objects in determining a good prefetch group size. Useful future work would include determining a model for the relationship between object size and prefetch group size.

5.4 Node marking: delayed vs. immediate

Chapter 4 showed that edge marking usually had a small performance advantage over node marking. This section briefly considers whether prefetching changes the relative performance of the two schemes. Edge marking has an advantage over node marking for objects that are fetched but never used, and one would expect to have more of this kind of object brought into the cache by any real (imperfect) prefetcher. However, it is possible to improve the performance of the node marking scheme by having it swizzle more lazily.

The node marking scheme described previously (hereafter called *immediate node marking*) swizzles each object when it arrives in the cache. The immediate node marking scheme will do unnecessary work for objects that are fetched but not used. However, it is possible to delay the swizzling of an object until after it has been fetched into the client cache. With this *delayed node marking* scheme, the cache manager makes an entry in the Resident Object Table for a newly-fetched object, but does not swizzle the pointers contained in the object. Instead, the object's method pointer is set to point to a dispatch vector containing pointers to special pieces of code SWIZZLE_i.

As with FETCH_i and FORWARD_i described in the previous chapter, there is an instance of SWIZZLE_i for each of the different legal indices into the method dispatch vector: attempting to execute the ith method of an unswizzled object instead causes SWIZZLE_i to be executed. Executing method SWIZZLE_i causes the following steps to take place:

1. Each pointer in the object is swizzled (creating a surrogate for the referenced object if necessary).

2. The object's method pointer is set to point to the appropriate dispatch vector for the class implementing the object (overwriting the pointer to a vector of SWIZZLE_i).

3. Method i is looked up in the real dispatch vector and executed.

Delayed node marking would be of little value if the prefetched object were sure to be useful. But as was shown in previous experiments in this chapter, real prefetching mechanisms have imperfect knowledge of the future computation, so they typically fetch several objects into the cache for each object that is actually used. If each object is fully swizzled as soon as it enters the cache, the system spends unnecessary effort swizzling objects that are never used. A delayed node marking mechanism ensures that relatively little effort is spent on prefetched but unused objects, which in turn allows more aggressive prefetching.

5.4.1 Hypotheses

Here are the three hypotheses tested in this section:

1. *Edge marking performs better in the presence of prefetching than immediate node marking*
   Since edge marking performs slightly better than immediate node marking in the absence of prefetching, and prefetching gives a slight advantage to edge marking, we expect edge marking to remain superior in the presence of prefetching. This hypothesis is supported by my experiments.

2. *Delayed node marking performs better in the presence of prefetching than immediate node marking*
   Since some prefetched objects will be unused, a delayed node marking scheme should perform better than an immediate node marking scheme.

3. *The performance of delayed node marking is comparable with that of edge marking*
   The work done for prefetched objects at fetch time by a delayed node marking scheme is not very different from the work done by edge marking, so the two should be roughly comparable in performance. This hypothesis is supported by my experiments.

5.4.2 Apparatus

The database used is as described in Chapter 4. The traversals used are the dense and sparse traversals, run with a cold cache, again as described in Chapter 4.

5.4.3 Results

Figure 5-15 shows the performance of edge marking, immediate node marking, and delayed node marking. These are compared for the bf-cutoff prefetcher.

For this prefetcher, delayed node marking performs better than immediate node marking on a sparse traversal, and it performs about the same as immediate node marking on a dense traversal. This is as expected: in a dense traversal most objects that are prefetched are used, while in a sparse traversal many prefetched objects are unused. The difference between the techniques is all on the client side, as can be seen from the fetches graph (middle) and storage graph (bottom). For all three techniques, the traversals fetch exactly the same objects, and so have identical fetching curves; but for each different technique, there is a different space cost and time cost. Edge marking consistently outperforms both forms of node marking.

For the perfect prefetcher (not shown), delayed node marking is always worse than immediate node marking, since the perfect prefetcher brings over exactly the objects needed. As a result there are never any unnecessary prefetches, and none of the cost associated with such an unnecessary prefetch. However, all other prefetchers make some errors in prefetching, and show much the same pattern as is shown for the breadth-first prefetcher, so I omit those graphs. For all measured prefetchers except a perfect prefetcher, delayed node marking comes close to the performance of edge marking; further, delayed node marking is much more effective than immediate node marking on the sparse traversal.

5.5 Related Work

Related work falls into three broad areas: clustering, system structure, and prefetching. Each of these is treated in one of the following subsections.

5.5.1 Clustering

Clustering is the problem of packing objects into pages so as to minimize page faults during the subsequent use of those objects. The roots of the techniques used are typically found in the techniques developed for packing records into pages [60, 75, 59, 49]. A significant amount of work has also been done on the specific problems of clustering objects into pages[6, 31, 67, 62, 15, 28]. Fetching a multi-object page instead of a single object is the only prefetching done in most object systems. Accordingly, for dynamic prefetching to be of any value, it has to do something better than can be done by clustering objects into pages. Clustering can be seen as a particularly static, heavily-constrained technique for computing object groups of fixed size.

There are relatively few good performance studies of different approaches to clustering. One is work by Stamos [67, 68], who compared the LOOM persistent object system to a paged virtual

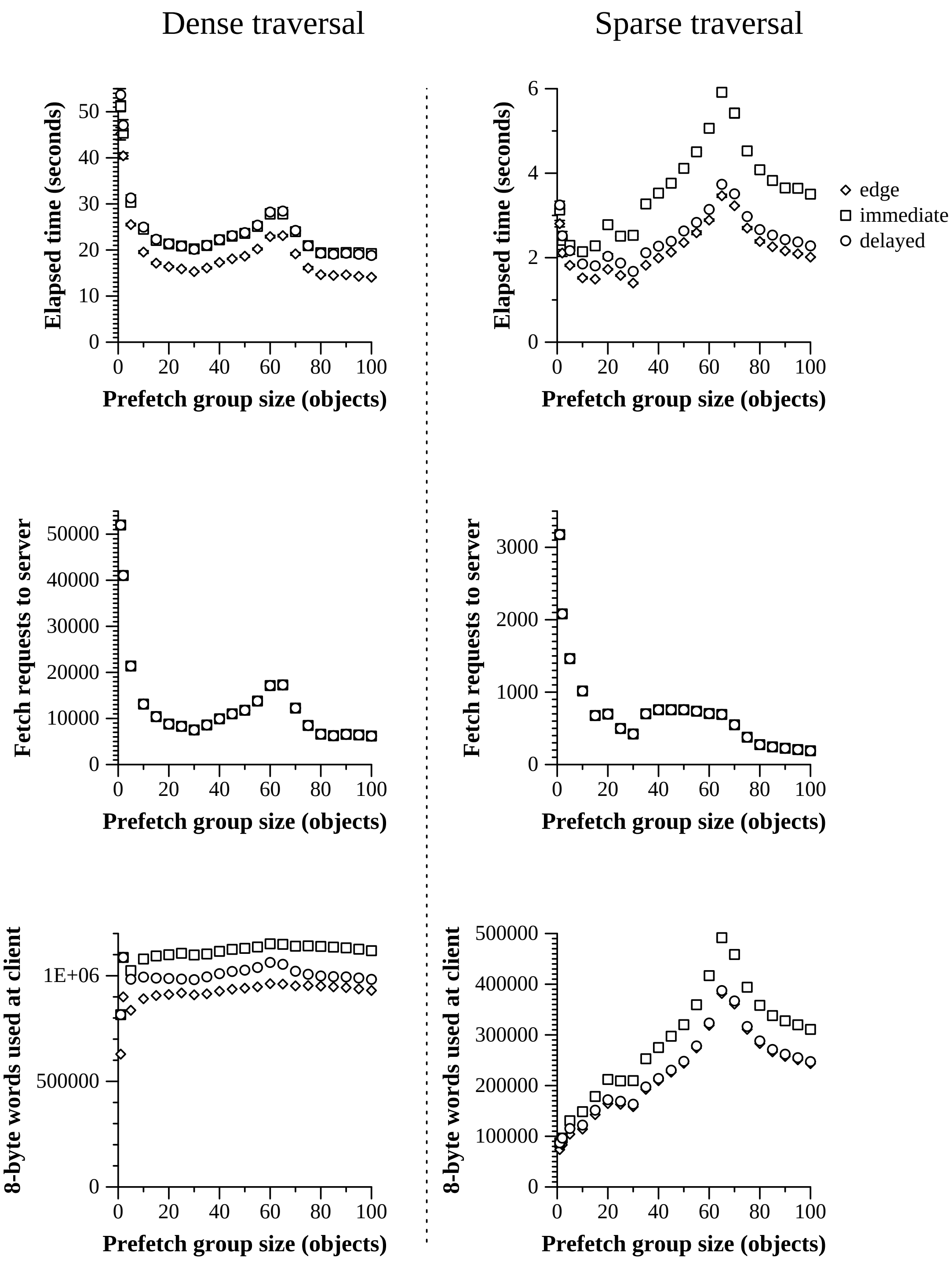

Figure 5-15: Edge marking, immediate node marking, and delayed node marking

memory so as to understand the tradeoffs between paging and object swapping. Stamos's work differs from my work on Thor in three ways. First, Stamos considers the problem of fetching from a disk, whereas this thesis deals with fetching objects from a server's memory. Second, Stamos considered only static grouping strategies (i.e. clustering, not prefetching). Third, Stamos considered a system with relatively small pages: 512-byte pages on a machine with 16-bit addresses.

Stamos concluded [67] that

- simple static clustering (such as depth-first or breadth-first based on static pointers) works better than a random arrangement of objects,
- more elaborate schemes do not work significantly better than simple schemes, and
- no simple scheme is clearly superior in all cases.

In a related technical report [68], Stamos also showed that LOOM's object fetching/replacement performs better for a small memory than a system using page fetching/replacement, although LOOM's advantage varies inversely with the quality of the initial placement (clustering). Stamos comments that the success of the paging system "depends on the existence and stability of quality initial placements."

The current Thor implementation runs on a network of DEC 3000 machines, which have 64-bit addresses. If every object on a 64-bit machine were simply 4 times larger than on a 16-bit machine, then Stamos's 512-byte pages would correspond to 2K-byte pages. However, the DEC 3000 actually has a page size of 8 Kbytes; in addition, objects do not all quadruple in size when addresses quadruple in size: character and integer fields of an object, for example, do not change in size. Thus, contemporary pages contain somewhere between 4 and 16 times the number of objects that were in Stamos's pages.

What is the significance of this increase in objects per page? As pages contain more objects, there is more work to be done each time a page is fetched (recall that the cost of concern is entering fetched objects into the Resident Object Table, and doing any necessary swizzling). If all pages continue to be full of relevant objects, this increased cost per page is not of concern: fewer pages will be fetched, with more work per page, for no net change in cost. However, it seems likely that object structure and grouping is determined by the semantics of an application, independent of the page size of the underlying implementation. It is reasonable to expect that there is a point at which the page size exceeds the size of logical object groups. As pages grow in size, the pages that contain some irrelevant objects become more important to performance. If the data stored has some kind of logical coherence at the granularity of (say) 10 objects, then one would expect to get a certain level of performance in a system where each page could hold a little more than 10 objects. If the same data were to be stored in a system where each page could hold about 100 objects, each page fetched could potentially involve an overhead of 90 irrelevant objects to be transmitted, copied, entered in tables, and/or swizzled. So as page sizes grow but logical object groups stay roughly the same size, the overhead of a page-fetching system grows, and a page-fetching system looks less attractive.

Stamos's work is rather old, and one might think that newer work in the area would show different results. However, Tsangaris and Naughton [71] recently compared the performance of a number of clustering techniques on a number of workloads. They clustered the data based on "training" workloads, then measured performance for "testing" workloads that might be quite different. They conclude that their stochastic clustering algorithm has the best performance across all workloads; unfortunately, stochastic clustering requires both estimating the parameters of a stochastic model from a trace of application references, and also mapping the objects into pages so as to minimize the probability that consecutive object accesses cross a page boundary. As a result, stochastic clustering is in Tsangaris and Naughton's terms "prohibitively computationally expensive to be applied directly in many situations." No inexpensive algorithm matches the performance of stochastic clustering across workloads. In addition, the highest-performing clustering algorithms are very sensitive to changes in the workload, whereas lower-performing ones are more stable. They conclude that "if stochastic clustering cannot be used, it is important to match an appropriate inexpensive clustering algorithm with a given application" and note that "the more precise the clustering algorithm, the more sensitive it is to mismatches between training and testing access patterns".

In short, clustering is far from a solved problem in object-oriented databases. In contrast to systems using clustering, prefetch group computation in Thor can happen either statically or dynamically, as described in this chapter. As was shown by the sparse and dense traversals, very different computations sharing the same objects can benefit from dynamic prefetching. There has been some work arguing for dynamic reclustering [15, 50]; however, such reclustering is expensive compared to dynamically computing prefetch groups (since reclustering involves disk writes) and it is challenging to decide when it is worth reclustering. A problem common to both clustering and dynamic prefetching is that it can be difficult to tune the system; at this point it is no easier to determine a good prefetch group size than to determine a good clustering. However, dynamic prefetching has the advantage that the performance of different applications can be tuned individually without affecting each other. In contrast, there is only a single clustering at any instant, and that clustering must be shared by all applications.

5.5.2 System Structure

Much previous work in prefetching for databases has assumed a centralized or single-site model, in which the executing application and the persistently-stored data are on the same machine. In a client-server system like Thor, the server's processor and memory are interposed between the application and the persistently-stored data. Accordingly, there are new possibilities for prefetching, since a processor using memory is much faster and more flexible than a disk controller and its disk; units larger or smaller than disk blocks can be fetched, and there is no cost advantage to prefetching units that are physically adjacent in storage, as long as the objects of interest are in the server's memory (as is assumed for this thesis).

DeWitt *et al.* carried out a study of client-server architectures [27] that compared an object server, a page server, and a file server. The different configurations were all derived from the WiSS system [18], which is organized as a layered structure. The different versions of the system correspond to placing the client/server split between different layers of the system.

DeWitt *et al.* found that no one architecture was superior in all situations. Their page server and file server did better for fetching than their object server, but the page server and file server were sensitive to whether they had a good clustering for the workload. The authors concluded that the "naive ideal" for reading would be to read pages of objects via NFS.

However, these conclusions require more examination, because the study's definition of object server seems problematic in two respects. The first problem with the study by DeWitt *et al.* is that it only considers an object server that fetches a single object at a time. The possibility of fetching multiple objects is dismissed on the grounds that it would make the server more complex. This is no doubt true for the experimental setup described; however, the argument is less compelling when considering building a system from scratch, rather than adapting an existing system like WiSS. After all, the complexity added by fetching multiple objects can be balanced against the complexity eliminated if the system does not use pages or page locking. I believe that DeWitt *et al.* have shown the need for fetching reasonably large groups of objects, but have not presented data that supports their conclusions about object-server architectures. More recent work [13] continues to make the case for pages by comparing them to single-object-fetching systems.

The second, less serious problem is that the authors define an "object server" to be something that is able to execute methods. This is confused: the granularity of entity shipped between server and client is orthogonal to whether methods can run at the server. Thor is an example of a system with servers that deal with objects, but do not run methods.

There is some other evidence previous to this work that points toward the possible value of dynamically computing object groups. Chang and Katz [15] argued for dynamic clustering and buffering at runtime based on structural information. In the context of a client/server system, their data and arguments can be viewed as supporting a model more like Thor's use of object groups than a typical page server system. Cheng and Hurson [17] determined that an object buffer pool and prefetching improved performance compared to a page store even when the prefetching was fairly sloppy (only 50% accurate). They describe a system with the page buffer and object buffer on the same machine, but there seems to be nothing that would preclude putting the object buffer on the client and leaving the page buffer on the server, as in Thor. It is worth noting that Cheng and Hurson earlier advocated an elaborate static clustering technique [16] to address the performance deficiencies of simpler clustering techniques. In their proposed clustering technique, objects were first clustered into primary clusters, then reclustered into secondary clusters. Their report on prefetching [17] partially repudiates the arguments they made in favor of their clustering scheme.

5.5.3 Prefetching

Early work on prefetching in databases, such as that by Smith [64], depends on repeated sequential access to data pages. Smith's prefetcher uses the length of the most recent group of sequential accesses (the *current run length*) to determine how many additional pages to prefetch. This use of sequentiality for prefetching is dependent on the data being laid out appropriately in consecutive pages. Smith's technique seems appropriate for disk transfers, where there is a strong incentive for sequential access to data; however, it is unable to detect or assist any non-sequential access, even if that access is repeated. Thor's prefetching is much more flexible than Smith's and similar work, since the units being prefetched are objects and they are being fetched from the server's memory rather than from disk.

Both the Fido cache [57] and optimal prefetching using data compression [72] learn arbitrary repeated access patterns, and are correspondingly more powerful than the mechanisms described in this chapter. Fido uses a neural-net mechanism whose details are unimportant for this discussion. The data compression prefetcher adapts the Lempel-Ziv compression algorithm [45] to prefetching, replacing a repeated sequence of page accesses with a single prefetch in much the same way that a data compressor replaces sequences of characters with a single code. While both these techniques are powerful and general, they have a significant start-up cost. If these techniques are used in a client/server system like Thor, they do nothing to address the cost of fetching objects before a pattern of use is apparent. If the objects fit into the client cache, then each is fetched only once: so by the time the system has collected the patterns needed to do good prefetching, the data of interest is at the client and no longer needs to be fetched. The only way to get any value from the state the prefetcher has built up is either to have the prefetcher shared with other clients or to save the prefetcher's state in the database in a form where it can be used subsequently (for example, in a stored prefetch hint called a crystal [21]). The prefetching mechanism in this thesis is a simpler approach, intended to boost the basic performance of the system without requiring the storage and tracking of prefetcher state.

Cheng and Hurson [17] proposed a model of a "moving window" for prefetching, with the window centered at the object currently in use and containing all of the objects reachable from that object. The next object access is likely to fall in this window, which moves smoothly around the object graph as different objects are used. This model is very similar to the dynamic prefetchers implemented in Thor and measured in this chapter. Cheng and Hurson use this model to motivate their approach, but claim that an exact implementation of the model is impractical. That assessment is correct for their purposes, but only because they are considering the client side as the prefetcher. The client knows the object being used, but it does not know the structure of the entire object graph, so it cannot know which extra objects to ask for. The key difference between their scheme and the Thor implementation is that, in Thor, the prefetcher is on the server side (i.e. it is a "presender"), where the structure of the object graph is known and everything in the moving window can be sent along with the current object. The prefetch groups sent in Thor often contain more objects than are in the moving window as defined by Cheng and Hurson, effectively containing some or all of the children's windows as well.

5.6 Conclusion

This chapter described the implementation and measurement of 7 prefetching techniques: pseudopaging, breadth-first, bf-continue, blind breadth-first, depth-first, class-hints, and perfect prefetching.

The data showed that the best compromise in the absence of information about the database and traversal was bf-cutoff: a simple breadth-first traversal that cuts off its exploration upon encountering an object already sent to the client.

It appears that with these workloads, a modest level of structural prefetching (prefetching no more than 30 objects for each object fetched) can give significant performance improvement (execution of the dense traversal in about 33% of the time for single-object fetching; execution of the sparse traversal in about 51% of the time for single-object fetching). The comparison to a page-fetching system is encouraging, although it does not represent a full comparison between the best possible object-fetching system and the best possible page-fetching system.

Experiments varying the structure of the database show that the average object size has more effect on the prefetching curve than does the number of objects in the database. Increasing the average size of objects requires that the prefetch group size be decreased to minimize elapsed time.

Chapter 6

Recovering Storage: Mechanism

Previous chapters discussed how to represent objects in the client cache, and how to fetch objects into that cache. This chapter and the next consider how to discard objects from the cache, recovering storage so that other objects can be fetched. This chapter describes the mechanism for recovering storage; the next chapter describes policies.

The client cache is managed so as to avoid writing objects back to the server except at transaction commit. This allows for a particularly clean, simple, and robust relationship between client and server. The client can voluntarily abort, or fail entirely, without contacting the server. The overhead incurred at a server unsure of a client's state is only the table recording the identities of objects or object groups sent to the client. In contrast, in a system that allows objects to be written back to the server, the server must manage both transient and persistent objects. Managing both transient and persistent data has the following negative effects:

- it adds to the complexity of the server, reducing its reliability;
- it requires the server to be more tightly coupled with the state of client transactions, again reducing the server's reliability; and
- it adds to the burden on the server, reducing the scalability of the system.

When objects are never written back except at transaction commit, cache management can encounter the problems described in this chapter as cache lockup. It is worth noting at the outset that cache lockup can always be solved by allowing objects to be written back to the server, although this solution requires accepting the problems outlined above.

There are two simple ways that one might manage the client cache. The first way, by analogy with a garbage-collected heap, is to use a garbage collector to reclaim the storage occupied by unreachable objects.

The second way, by analogy with virtual memory, is to replace objects: when space is needed for new objects, some "old" objects are chosen by some suitable algorithm. The chosen objects are evicted from the cache, and their space is used for new objects. However, this simple replacement approach does not work in Thor.

Recall from Chapter 4 that Thor uses direct swizzling. In a system using direct swizzling, an inter-object reference may be represented by the referenced object's address in the client cache. Because objects in such a system may point directly to each other, the simple replacement described above does not work, because of the possibility of dangling references; this problem is explained below. This chapter describes a mechanism called *shrinking* that is derived from replacement. Shrinking requires no special hardware support, but does work in a system with direct swizzling.

This chapter shows that neither garbage collection nor shrinking is a sufficient technique on its own. With either technique, it is straightforward to construct examples where it is impossible to recover enough storage to finish a computation, even though the client cache is more than large enough to hold the objects needed by the client computation at any one time. This problem is called *cache lockup*. As noted above, cache lockup can be remedied by allowing objects to be written back to the server, but for this thesis I assume a structure in which objects cannot be written back to the server except at transaction commit.

The first contribution of this chapter is identifying the problem of cache lockup and the corresponding inadequacy of these two cache management techniques in isolation. The second contribution is pointing out that shrinking and garbage collection are complementary: each is able to repair the condition that causes the other to fail. Accordingly, a combined mechanism of garbage collection and shrinking should be less likely to suffer cache lockup. The third contribution is using the implemented Thor system to demonstrate the reality of cache lockup and to confirm that a combined mechanism effectively eliminates cache lockup.

The next section describes in more detail why simple replacement does not work in Thor, and why shrinking is necessary instead. Section 6.2 considers the possibility of using shrinking to manage the Thor client cache. Section 6.3 considers the possibility of using garbage collection to manage the Thor client cache. Section 6.4 explains how garbage collection and shrinking are complementary, and should work better together than either on its own. Section 6.5 reports results from an experiment showing that a combined mechanism works much better than either mechanism on its own. Section 6.6 summarizes related work.

6.1 Shrinking Instead of Replacement

As mentioned above, Thor uses direct swizzling, so an object in the cache may contain a pointer directly to another object in the cache. Figure 6.1 shows an object A that points directly to object B.

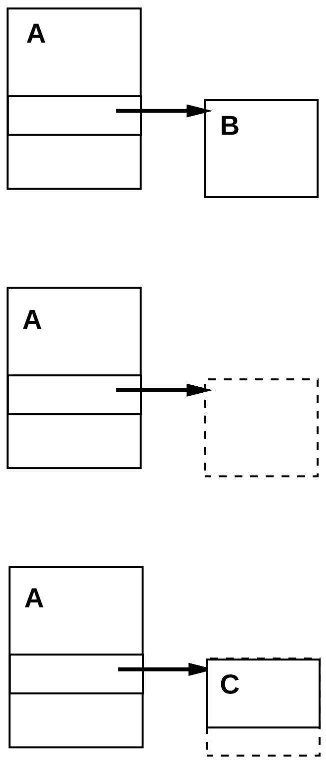

Figure 6-1: Incorrect reuse of storage

```
loop {
  accept client call
  perform work for client call
  if (space left < gc trigger) {
    run gc
    if ((space recovered by gc / halfspace size) < shrink trigger) {
      shrink objects up to shrink fraction
      run gc
    }
  }
}
```

Figure 6-2: How shrinking is triggered

As a result, the cache manager cannot simply evict B and reuse its storage for a different object C. If such eviction were allowed, object A could subsequently dereference its pointer and incorrectly access object C instead of object B, simply because because C happens to occupy the same storage as the evicted object B. The comparable problem in a virtual memory system is a reference to an evicted page, which is handled by hardware. Although it is tempting to use such page-faulting hardware to assist object systems [42, 63] some evidence [34] suggests that such page-faulting hardware is ill-suited to support object operations, primarily because of the mismatch between the size of a typical object and the size of a typical page.

To avoid the dangling-reference problem using only software, objects are not simply evicted and replaced in Thor. Instead, an object is removed from the cache by being converted to a surrogate (surrogates were described in Chapter 4). Because a surrogate is always smaller than an object, this mechanism is called *shrinking*.

6.1.1 When shrinking happens

In the current implementation of Thor, shrinking is triggered only after a garbage collection has completed; see Figure 6-2 for a pseudocode description of triggering. If the garbage collector fails to free a specified fraction of all possible free space, then shrinking takes place, followed by another garbage collection that compacts the remaining objects so as to recover the space freed by shrinking.

6.1.2 What Happens During Shrinking

The *shrink fraction* is a parameter that determines the fraction of objects to be shrunk whenever shrinking is triggered. For this thesis, the shrink fraction is constant for the duration of a computation. The shrink fraction only affects the RANDOM, CLOCK, and LRU policies.

An object can be shrunk only if it is persistent and unmodified. As a simplification, this thesis deals only with read-only computations, in which all persistent objects are unmodified, and there are no volatile objects. Accordingly, all objects are shrinkable.

Shrinking an object is quite cheap. Shrinking requires only that the first two words of the object be overwritten by a surrogate for the object. The first word of the surrogate is a pointer to surrogate code; the second word contains the name of the object.

Shrinking would not be necessary if it were possible to find and fix up all of the pointers like the one contained in object A. One way to find all such pointers is to keep backward pointers for each pointer in the system. Such an approach seems to entail too much overhead to be of interest; Section 6.6 considers the virtues and limits of this approach in more detail. A more practical way to find relevant pointers is to use a mechanism like a garbage collector that traces all reachable pointers from designated roots. Section 6.4 returns to this possibility.

6.1.3 Unshrinkable objects

In a typical computation, only a small fraction of objects touched are modified, so that relatively few objects are unshrinkable. Nevertheless, it is useful to digress and sketch how the presence of unshrinkable objects is likely to affect performance.

As long as the fraction of unshrinkable objects is *smaller* than the shrink fraction, unshrinkable objects should have a rather small effect. There are two effects caused by unshrinkable objects. First, it becomes more difficult to find shrinkable objects; however, the cost of shrinking is relatively small compared to the cost of fetching and garbage collection. Second, the effective size of the cache is smaller because of the objects that must be retained.

If the fraction of unshrinkable objects is *larger* than the shrink fraction, unshrinkable objects have an effect in addition to the two described above. In such a configuration, objects are copied by the garbage collector that would otherwise have been shrunk; this represents an additional cost for each garbage collection.

The presence of a large number of reachable, volatile objects could change the behavior of the system quite dramatically. If it were common for a Thor application to create large numbers of volatile objects that survived garbage collection, it would be appropriate to build a separate cache for those volatile objects. This would correspond roughly to the "nursery" or "first-generation" space used by generational garbage collection schemes. The schemes described in this chapter would then be applied only to the cache of permanent objects. An object that became persistent would have to be copied from the volatile cache to the permanent cache.

6.1.4 Garbage collection and prefetching

As described in Chapter 5, the prefetcher at the server may use information about the objects that are present at each client when making prefetching decisions. If an object is sent by the server and subsequently shrunk by the client, the client must inform the server of the change. If the client does not inform the server appropriately, that object will not be prefetched in any subsequent fetch from that server. Each time the prefetcher at the server considers sending that object, it will decide (incorrectly) that the object is already present at the client. Thus, the server will not send that object until it is explicitly fetched. Failing to keep the server up-to-date can cause the system's performance to degrade to the point where it behaves as though there were no prefetching, only single-object fetching. In addition, good performance of the system's concurrency control mechanism [1] and distributed garbage collector [48] also depend on having information at the server that is close to the actual state of the client.

Correctness does not require updating of the server's information about the client state; the updating is intended to improve performance. Instead of updating the server every time an object is shrunk, it is possible to delay updating the server until a number of shrinks have occurred, amortizing the cost of communication with the server. In the implementation measured here, the server is updated after each client cache garbage collection. After a garbage collection, the client knows precisely which objects are in use in its cache. As part of the process of rebuilding the resident object table, information is sent to the server to update its model of which objects are present at the client.

6.2 Pure Shrinking

Can Thor's client cache management work entirely in terms of shrinking? No. Although the cache may be more than large enough to hold all of the objects that need to be in the cache simultaneously, there are two ways that the cache can fail to function. First, the cache can reach a state where it is full of surrogates, none of which can be discarded because they may be reachable from elsewhere in the cache. Second, the cache can become so fragmented that there is no longer any contiguous piece of storage large enough for an object that has been fetched.

Here is a simple example of a computation that fails when using only shrinking: all of the objects used are the same size, and there are too many to fit all of them in the cache at once. Shrinking an object requires leaving a surrogate behind in case other objects refer to that object. Because of the space occupied by the surrogate, the space recovered from shrinking the object is less than the space required for the object being fetched. As a result, shrinking an object does not free up enough space to store a new object of the same size as the one shrunk. Shrinking more than one object does not help, since each surrogate must occupy the beginning part of storage that was occupied by the object it replaces. So although shrinking two objects recovers enough bytes to store a new object, those bytes are split into two pieces, neither of which is large enough to contain the whole object. There is no way to find enough free space using only shrinking. This state is a form of cache lockup.

6.3 Pure Garbage Collection

Can Thor's client cache management work entirely in terms of garbage collection? No. A garbage collector recovers storage occupied by objects that are unreachable. In Thor and similar systems, the client cache contains only a portion of a much larger persistent heap. It is quite likely that the cache is full of reachable objects. An object is *visible* to the client application if it has been returned as the value of some computation and the application has not explicitly indicated that it is willing to forego using that object. The client application may invoke an operation of any visible object. Since the computation might navigate to any of the objects reachable from any visible object, each visible object must be treated as a root for garbage collection. This is a simple and safe interface, but it means that it is easy for a client application to hold a collection of roots such that everything in the client cache is reachable.

A straightforward implementation of garbage collection in the client cache will fail if all of the objects in the cache are reachable. As with pure shrinking, cache lockup occurs: the computation is unable to proceed because all of the objects in the cache must stay in the cache, meaning that there is no room for any new objects. In a conventional program, exhausting the heap is indeed catastrophic and there is neither a need nor a means to recover. However, in Thor it is possible to recover from a cache lockup by shrinking the local copies of some unmodified persistent objects.

6.4 Combining Shrinking and Garbage Collection

To summarize pure shrinking and pure garbage collection:

- pure shrinking can be based on tracking usage patterns, but has no reachability information; as described, it can shrink old objects but cannot discard unreachable ones. It eventually fails as the cache fills with surrogates or becomes too fragmented to find storage for newly-fetched objects.

- conventional garbage collection is based on reachability, but has no information about usage patterns; as described, it can discard unreachable objects but cannot shrink old ones. It eventually fails as the cache fills with reachable but irrelevant objects.

Phrased in this way, it is apparent that shrinking and garbage collection are complementary storage management techniques; each can fix the problem that causes the other to fail.

In addition to the performance problems previously noted, reference counting systems are usually unable to reclaim cyclic structures. However, reference counting combined with shrinking would suffer less from that problem. All cycles containing at least one unmodified persistent object would eventually be broken, because that object would eventually be shrunk to a surrogate. A surrogate contains no references to other objects in the cache; so replacing an object with a surrogate (shrinking the object) breaks any cycle of which that object was a member. Figure 6-3 shows a cycle of objects, and each object is labelled with its reference count. Shrinking one object to a surrogate breaks the cycle, and allows the storage to be reclaimed. However, even with this improvement, a reference counting scheme seems likely to be inferior to a copying garbage collector. A copying collector allows allocation of storage to be very cheap, and allows storage to be compacted.

6.5 Demonstrating cache lockup

The previous section argued that a combination of garbage collection and shrinking is needed to manage an object cache. This section tests that argument with the running Thor system, by determining the smallest client cache in which a particular computation can run without experiencing cache lockup. Different configurations are compared in terms of *cache lockup point*. The cache lockup point for an application and configuration is the largest cache size for which the traversal fails to complete due to an inability to allocate cache. Thus, a small number is desirable.

Previous chapters have already established the (very slight) superiority of edge marking to node marking in a system without cache management, both with and without prefetching. This chapter confirms the superiority of edge marking when storage must be recovered. However, the performance difference between node marking and edge marking is very small when compared to the difference between a system using shrinking and an otherwise-identical system without shrinking.

6.5.1 Hypotheses

Here are the hypotheses to be tested:

1. *When using garbage collection only, edge marking has a lower cache lockup point than node marking.* From simulations, I expected that edge marking would do slightly better than node marking at avoiding cache lockup when using only garbage collection. This hypothesis is supported by my experiment.

2. *Shrinking combined with garbage collection has a lower cache lockup point than either shrinking or garbage collection individually.* As argued earlier in this chapter, shrinking and garbage

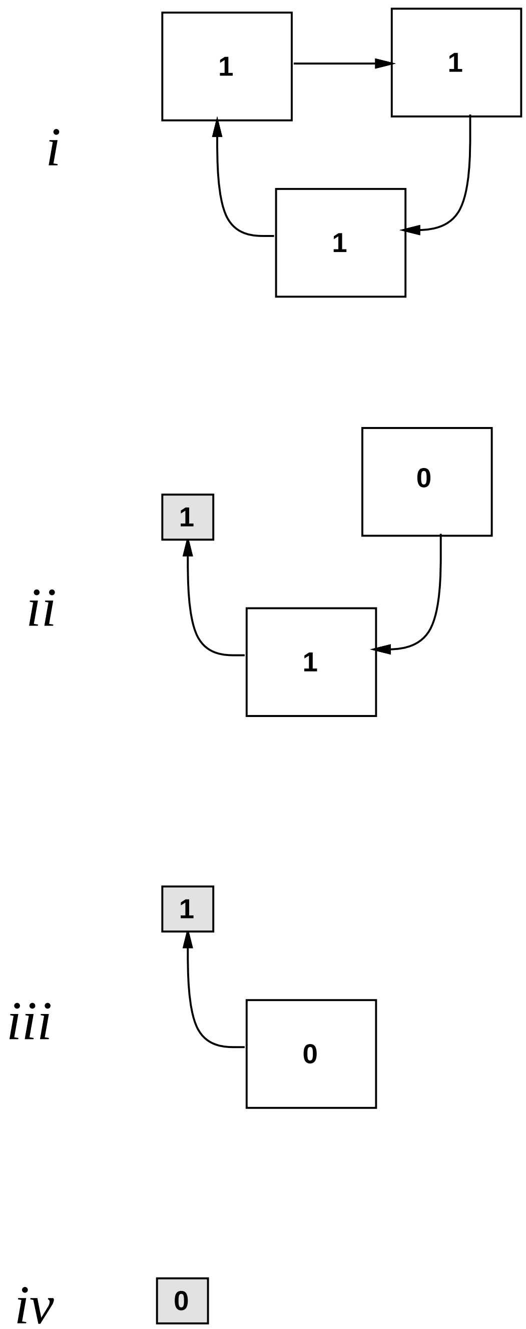

Figure 6-3: How shrinking breaks cycles

collection should have complementary strengths for recovering storage. This hypothesis is also supported by my experiment.

6.5.2 Experimental setup

The database used is the same as the one described in Chapter 4. The traversal used is the dense traversal, run on a cold cache, also as described in Chapter 4. However, the transfer of control is different from that described previously. 4. To explain the difference, this section starts by sketching the implementation of the garbage collector in the current Thor system and the rationale for that implementation.

Thor is intended to be portable to a wide variety of platforms. It is difficult to identify roots at arbitrary points in the execution of a computation if the register allocation and stack organization are controlled by a compiler provided by some third party. Rather than making the collector dependent on details of particular compilers, the current Thor client garbage collector may run only when a client call ends. When control returns to the client, the stack for execution of database operations is empty, and so there is no possibility that a frame on that stack contains a reference that should be treated as a root for garbage collection. When storage management is an issue, as it is for this chapter and the next, the computation must take place in operations of sufficiently small granularity that the garbage collector has a chance to run before storage is exhausted. Accordingly, whereas the previous chapter's experiments essentially ran the entire traversal as a single database operation, this chapter and next traverse only one whole CompositePart (and its component AtomicParts and Connections) as a single database operation.

Three configurations are compared. None of the configurations use prefetching. Prefetching would tend to induce cache lockup sooner; this experiment is intended to find the smallest cache that works, not the fastest completion of the traversal. Two configurations use only garbage collection to manage the cache: one uses node marking and one uses edge marking. The third configuration uses node marking and shrinking in an arbitrarily-chosen combination; the next chapter considers how best to combine shrinking and garbage collection. For this experiment, 50% of the objects, chosen at random, were shrunk to surrogates whenever the garbage collector was able to recover less than 5% of the cache.

6.5.3 Results

Figure 6-4 shows cache lockup points for six configurations: both edge marking and node marking in combination with shrinking only, garbage collection only, or both shrinking and garbage collection. For each configuration, the larger light bar (labeled "total") shows the amount of storage that is occupied at the end of the traversal if there is enough cache available so that no storage needs to be recovered. The "total" bar's height is the same as the amount of storage used with a prefetch

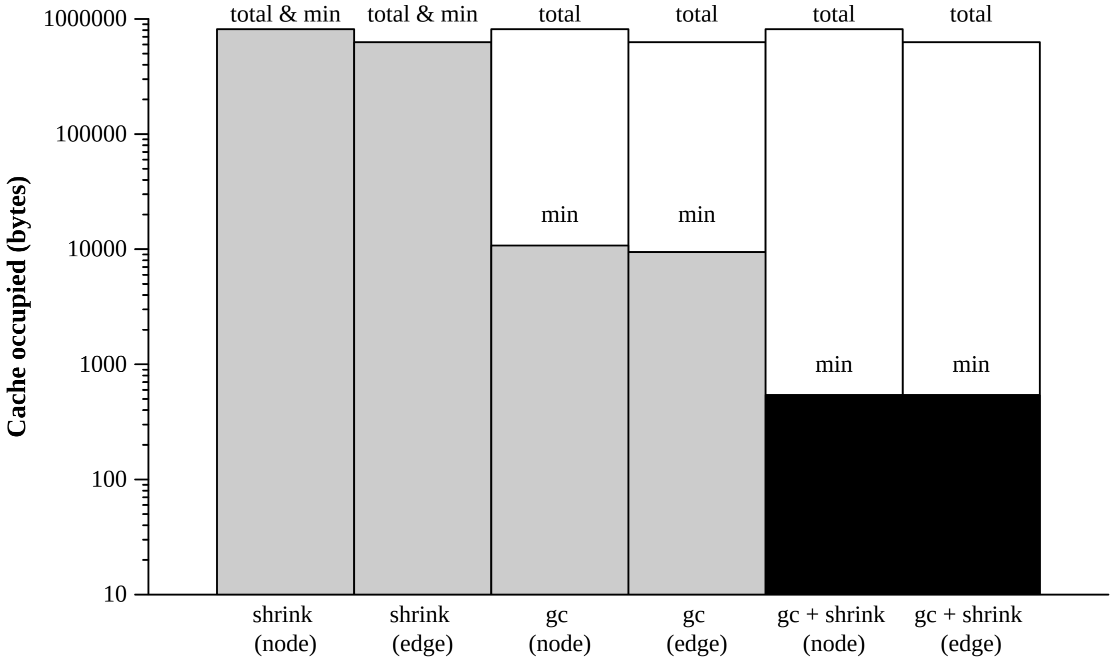

Figure 6-4: Cache Lockup Point

group size of 1 in the previous chapter. The smaller dark bar (labeled "min") represents the cache lockup point. Thus, the difference between the top of the light bar and the top of the dark bar represents the best "compression" of the computation into a smaller cache that can be achieved by each configuration. Note that the vertical axis is logarithmic. For this workload, a system using shrinking only locks up as soon as any cache storage needs to be recovered, so there is no difference between the "total" bar and the "min" bar.

From the graph, it is apparent that edge marking is slightly superior to node marking. This is to be expected, since the edge marking system does not create surrogates that consume extra space in the cache. However, both systems lock up at roughly the same point.

For the combined mechanism, the lockup point is about an order of magnitude lower. In fact, this lockup point is determined by the amount of storage required before the first opportunity to run the garbage collector (just before the first operation of the benchmark is executed). For the implementation of Thor measured, this cache lockup point represents the minimum storage required. The combination of garbage collection and shrinking has eliminated the problem of cache lockup.

However, at this minimum cache, the traversal takes nearly 6 times as long as it does when no cache management is required. For most applications, it is important to understand how performance degrades before the system stops completely. The next chapter considers details of when and how shrinking should take place for good performance.

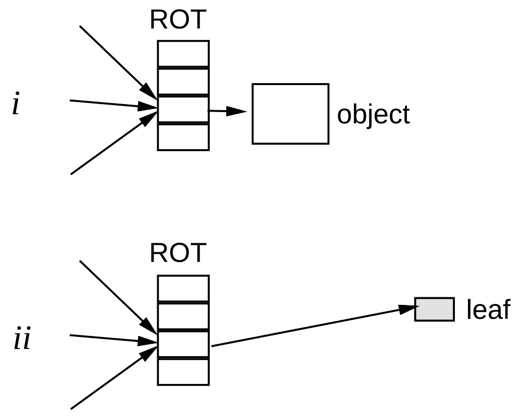

Figure 6-5: Contracting an object in LOOM

6.6 Related Work

6.6.1 Pure Shrinking

LOOM [39] was the first system with mechanisms similar to the swizzling and surrogates presented in this thesis. Leaves in LOOM are somewhat like surrogates: a leaf is a small data structure containing the name of an absent object, and a LOOM object can be "contracted" into a leaf, in a way that is roughly similar to shrinking. However, LOOM uses *indirect* swizzling; a LOOM leaf is more like an optional field of the Resident Object Table (ROT) than like an object in the cache, since its actual address in memory is irrelevant. Because of the indirection through the ROT in LOOM, there is always exactly one reference to an object or leaf that needs to be changed if the object is shrunk or fetched. The indirection through the ROT also means that it is straightforward to relocate an object within memory. Several other object-oriented databases (GemStone[8, 66], Orion[41], Jasmine[36]) use basically the same cache structure as LOOM. We chose to use direct swizzling in Thor to avoid the time and space overheads caused by indirect swizzling. Those overheads are similar to the overheads involved in the difference between node and edge marking; knowing what we now know about the small difference in that case, we might choose a LOOM-style approach instead of direct swizzling if we had an opportunity to reimplement Thor's client cache.

Figure 6-5(i) shows how every inter-object reference in LOOM goes through some entry of the ROT. In 6-5(ii), that object has been contracted to a leaf, but the leaf can be elsewhere in memory. Figure 6-6 shows how a Thor surrogate, by contrast, actually takes the place of an object, and must occupy the same starting position in memory as the object it replaces.

O_2 uses client virtual memory to manage memory. If there is no more swap space, space for objects "is freed"; the exact process is unclear [26]. It appears that no attempt is made to manage storage until swap space runs out; this means that the client can be thrashing with reads and

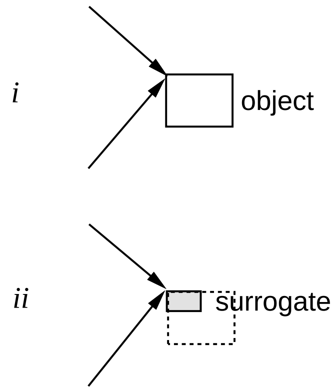

Figure 6-6: Shrinking an object in Thor

writes to its local disk, in addition to whatever costs are incurred from the server's disk. Thor's design proceeds from the idea that the database system should be controlling whatever disk accesses take place, so that the client cache does not depend on having a virtual memory system available. Although O_2 uses indirect swizzling, so that moving an object in the address space is relatively easy, the address of an O_2 object never changes while the object is in memory; there is no copying garbage collection.

Kemper and Kossmann [40] studied a mechanism of *reverse reference lists* for doing storage management without using a garbage collector. Each object keeps a list of the objects that hold references to it. When an object needs to be evicted from the cache, its reverse reference list makes it simple to find and fix up those objects whose references might be incorrect. However, this technique involves a considerable overhead in terms of storage required, and the maintenance of the lists is quite expensive. Any mutation that changes the object graph requires changing one or more reverse reference lists. The reverse reference lists must be kept up-to-date at all times in case storage management is needed; this represents a considerable waste of resources in situations where there is adequate storage and the cache manager does not need to evict objects. As previously noted, reference counting garbage collectors have trouble competing with copying garbage collectors [74]. A reference counting system only requires a counter per object, and only needs to increment or decrement a counter at each object graph mutation; in contrast, a system using reverse reference lists must store a potentially large bag of reverse references for each object, and must insert addresses into these bags or remove pointers from these bags at each such mutation.

6.6.2 Pure Garbage Collection

LOOM [39] built on an existing Smalltalk system, so instead of using a garbage collector it extended the reference counting scheme used for that system. Because of the use of reference counting (which does not compact of storage), allocation was considerably more complex and costly in LOOM than it is in Thor. In a system with compaction, an allocation requires only changing the freespace pointer. Wilson's survey [74] concludes that reference counting systems have generally performed poorly compared to copying collectors, and there is nothing in the structure of Thor that would contradict that assessment.

Alltalk [69] uses a mechanism that integrates garbage collection with the pushing and popping of frames as Smalltalk methods are called and return. This mechanism requires more control over the execution stack than Thor has. For portability, Thor methods are ultimately compiled by the local C compiler, rather than being compiled directly to native code.

Butler's work [7] is concerned with garbage collection for an object-oriented database, but is primarily concerned with the problem of garbage collecting persistent storage. In a client/server system like Thor, the client cache management problem is decoupled from the problem of managing the server's persistent storage.

6.6.3 Shrinking and Garbage Collection

Cooper *et al.* describe how they modified Standard ML of New Jersey (SML) [51] and implemented an SML-specific external pager for Mach so that VM cache management could take advantage of the knowledge of the SML garbage collector [19]. The garbage collector marked pages as discardable or nondiscardable based on their current role in the garbage collection process. The combination of shrinking and garbage collection in Thor has some of the same character as their system, but Thor applies the idea to individual objects instead of pages, and to a client object cache instead of a VM cache.

O'Toole [56] has sketched a scheme for using replication garbage collection and shrinking (which he calls *re-surrogating*) to manage a cache of persistent objects. The garbage collector currently implemented in Thor is a simple copying collector, which appears to be less flexible than a Nettles-O'Toole replicating collector would be in choosing when to start shrinking objects. Interesting future work would be to implement a replicating collector for Thor and evaluate it experimentally.

Chapter 7

Recovering Storage: Policies

The previous chapter demonstrated that freeing storage in the client cache requires both shrinking and garbage collection. The policy used for that demonstration was arbitrarily-chosen and — as it turns out — not especially good for completing the benchmark quickly. This chapter explores what makes for a good shrinking policy.

The chapter explores how the Thor implementation behaves with seven different shrinking policies. Four of these (RANDOM, CLOCK, LRU, and TOTAL) are realistic: they could be used for real applications. The details of these policies are described later in the chapter. The other three policies are unlikely or impossible policies that show interesting effects: CLEARCACHE shows the cost of failing to cache objects between transactions when consecutive transactions share some objects; MINFETCH shows how good shrinking could ever be; and MAXFETCH shows how bad shrinking could ever be.

Shrinking policy is an important choice for a system and application, but the experiments in this chapter show that it is not the most important factor. Both the prefetching policy and the maximum prefetch group size are *more* important for performance than is the shrinking policy.

The chapter starts with an overview of the experimental technique. Section 7.1 describes the *wide database* and *shifting traversal* used. Section 7.2 describes the realistic shrinking policies being compared, and the intuition supporting each. Section 7.3 presents hypotheses to be tested by experiments. Section 7.4 explains the data collected for a single shrinking policy (TOTAL) in a single configuration. Section 7.5 shows the effect of varying parameters affecting shrinking performance while keeping one shrinking policy (still TOTAL). This serves as context for Section 7.6, which compares the various shrinking policies. Finally, Section 7.7 summarizes related work, and Section 7.8 offers a summary and conclusions for the chapter.

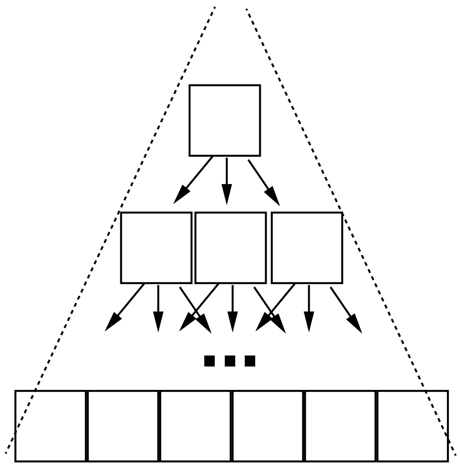

Figure 7-1: A single database

7.1 Shifting Traversal and Wide Database

For the experiments reported in this chapter, I adapted one of the OO7 benchmarks to provide more interesting and realistic cache behavior. The *shifting traversal* is a new traversal. It is based on the dense traversal used in earlier tests (see Chapter 4 for details). The shifting traversal consists of multiple partially-overlapping iterations of the dense traversal. This realizes, although in a very simple form, the phase/transition behavior that characterizes real programs [25]. Each of the dense traversals is henceforth called an *elementary* traversal. The complete shifting traversal has a shifting working set [23]: with a large enough cache, each elementary traversal except the first has 2/3 of its objects present from the previous traversal.

The shifting traversal uses a *wide database* that is approximately two of the previously-used databases, joined at the roots. Figure 7-1 shows a schematic representation of a single database, with structure similar to the database used in the previous chapters. The root of the database is at the top.

Figure 7-2 shows a schematic representation of the wide database used in this chapter. The root Assembly has 6 children instead of the usual 3. Effectively, there are two similarly-structured databases sharing a common root.

Figure 7-3 shows the structure of the traversals. Each child of the root is labelled with the number(s) of the elementary traversal(s) in which it is used. There are four elementary traversals. The first elementary traversal uses the children labelled 1, the second uses the children labelled 2, and so on. Each elementary traversal uses the leftmost of its children first and proceeds rightward. Any given elementary traversal uses exactly three of the six children of the root. Each successive elementary traversal (except the first) uses two of the three children used previously, and also uses

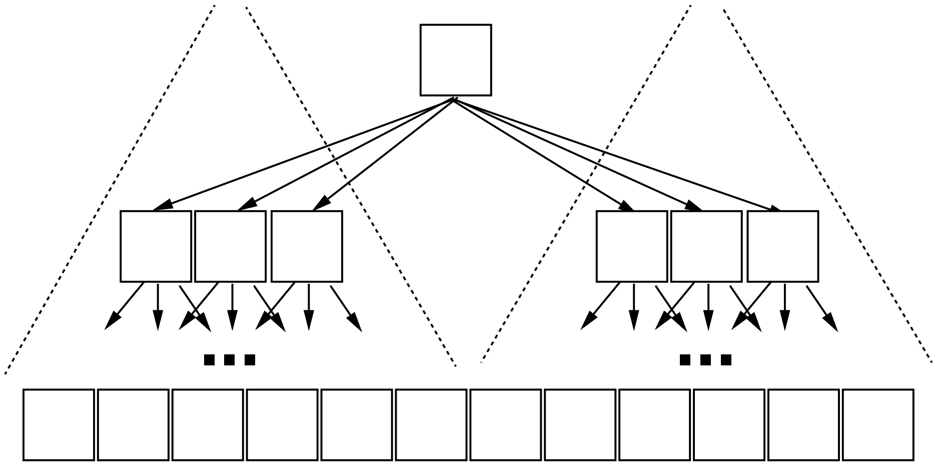

Figure 7-2: The wide database

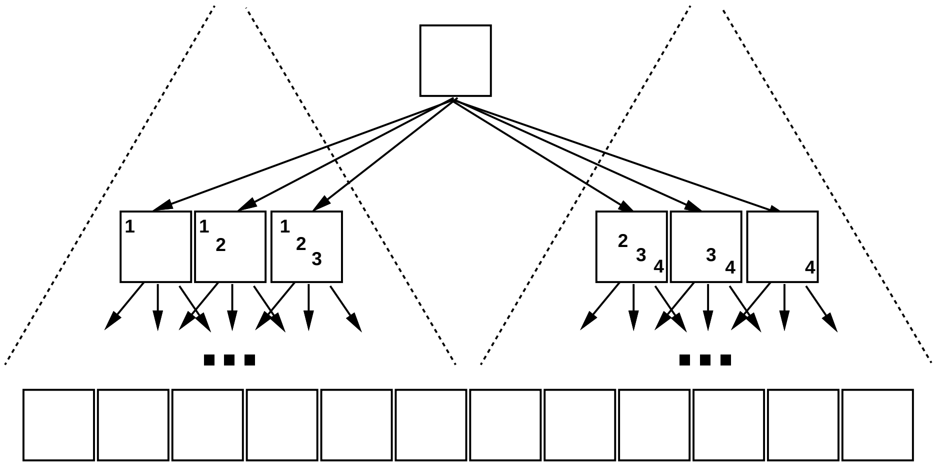

Figure 7-3: Traversals of the wide database

part	size	bytes (each)	number	bytes (total)
ComplexAssembly	$h + 4f + v_3$	168	13	2184
BaseAssembly	$h + 5f + v_3$	176	27	4752
CompositePart	$h + 7f + v_1 + v_{20}$	392	81	31752
AtomicPart	$h + 7f + 2v_3$	272	1620	440640
Connection	$h + 4f$	88	4860	427680

Figure 7-4: Sizes of objects in each subtree

variable	meaning	size (bytes)
h	Header and info overhead	56
f	Field	8
v_1	1-element vector	64
v_3	3-element vector	80
v_{20}	20-element vector	216

Figure 7-5: Sizes of object components

one child that was not used before.

In contrast to the databases used in previous chapters, the wide database is generated in a way that guarantees there is no sharing of CompositeParts among BaseAssemblies. Such sharing would interfere with the intended shifting of the working set.

The size of the working set for most of the computation is the storage required for two subtrees of the database. (Although each elementary traversal uses *three* subtrees, two successive elementary traversals share only *two* subtrees.) Failing to keep those two shared subtrees in memory between traversals will cause extra fetches to occur. Each subtree contains 13 ComplexAssemblies, 27 BaseAssemblies, and 81 CompositeParts. Each CompositePart contains 20 AtomicParts and 60 Connections. Figure 7-4 shows the space taken by the various objects in the database. Figure 7-5 (which repeats Figure 3-7) explains the space costs of the various components of objects.

Each subtree occupies about 907000 bytes, or about 885 Kbytes. Two subtrees occupy about 1770 Kbytes. Since the current implementation of Thor uses a copying collector, the smallest cache that contains the working set must be at least twice this size, or about 3500 Kbytes. It is not useful to measure the performance of the system with a cache of 3500 Kbytes or less, since it represents thrashing. Experiments confirmed this calculation: performance of any configuration decays quickly as the cache size shrinks to a size near or below 3500 Kbytes.

For all these experiments, the swizzling used is edge marking (see Chapter 4 for details). There are 254180 pointer tests in the dense shifting traversal. With no prefetching, 49572 of these tests cause objects to be fetched, for a miss rate of 19.5%. Most of the configurations measured (with a maximum prefetch group size of 30 objects) have a base miss rate of about 3.5%, meaning that roughly one in 28 attempts to use some field of some object causes that object to be fetched from the server.

7.2 Overview of Policies

This section describes each of the realistic policies (TOTAL, LRU, CLOCK, and RANDOM) measured by experiments, and provides some intuition for why each might be a useful shrinking policy.

7.2.1 TOTAL

The TOTAL policy shrinks every shrinkable object. It is simple and reliable, but we would expect that it almost always shrinks too many objects. Since the server can send extra objects with each object fetched (as described in Chapter 5), the effect of shrinking too many objects may not be too severe. The TOTAL policy has the advantage that it guarantees that no cache space is being occupied for objects that are no longer useful.

7.2.2 RANDOM

The RANDOM policy simply takes all of the shrinkable objects in the cache and with some probability (computed from the shrink fraction) decides randomly whether to shrink each one. Like TOTAL, RANDOM does not require any information about object usage to be maintained, and so avoids the space and time costs required for that maintenance. For hardware caches, a random replacement policy is approximately as good as more elaborate policies [32]. It is possible that the same would be true for a client object cache; perhaps there is not enough locality in the pattern of references to justify any policy more elaborate than picking some victim and evicting it.

7.2.3 CLOCK and LRU

Shrinking has some similarities to page replacement in virtual memory systems. Many virtual memory systems use an approximation of a Least Recently Used (LRU) policy for page replacement. The shrinking policies called LRU and CLOCK are both based on a least-recently-used scheme, but differ in how closely they approximate a true LRU system. Essentially, the LRU policy is a very close approximation, while CLOCK is only a coarse one-bit approximation, similar to what is commonly used for virtual memory systems (e.g. [44]). Either policy is plausible if computations on object databases display locality of reference similar to what is observed in virtual memory systems.

7.3 Hypotheses

These hypotheses about recovering storage in Thor are tested by experiments described in this chapter:

- *Inter-transaction caching performs better than caching only within a transaction.* If there is some overlap of objects between transactions, as there is with the elementary traversals of the shifting traversal, one would expect better performance from a cache management technique that allowed objects to stay in the cache between transactions, in contrast to systems that clear the cache at the end of a transaction. This hypothesis is supported by my experiments.

- *Garbage collection and shrinking provide good performance.* One might expect that a combination of garbage collection and judicious shrinking when needed would allow a traversal to slow down gracefully as the cache size is reduced. This hypothesis is supported by my experiments.

- CLOCK *shrinking is superior to* RANDOM *shrinking.* By analogy to virtual memory page replacement, one might expect the CLOCK policy to perform better than RANDOM shrinking. This hypothesis is supported, but only weakly.

- LRU *shrinking is superior to* CLOCK *shrinking.* Since the LRU shrinking technique can make finer distinctions as to which objects were recently used, one might expect it to perform better than CLOCK shrinking. This hypothesis is supported by my experiments.

- *Only a small fraction of the objects in the cache should be shrunk.* One might expect that carefully shrinking a few objects when needed would maximize performance. This hypothesis is supported by my experiments, but needs qualification: shrinking too few objects is more expensive than shrinking too many. If it is difficult to establish the best operating region, it is better to shrink too many objects than to shrink too few.

- MINFETCH *shrinking is superior to all other shrinking.* MINFETCH is designed to minimize the number of fetches (the details of the algorithm appear in Appendix A). However, it is important to check whether minimizing fetches is enough to minimize elapsed time. This hypothesis is supported by my experiments.

7.4 Understanding the Data for One Policy

Before comparing shrinking policies or varying the parameters of the configuration, it is important to understand the data being presented for a single policy in a single configuration. Figure 7-6 shows the behavior of the TOTAL shrinking policy in its default configuration. The details of the default configuration are described in the next section, which considers varying the parameters. The

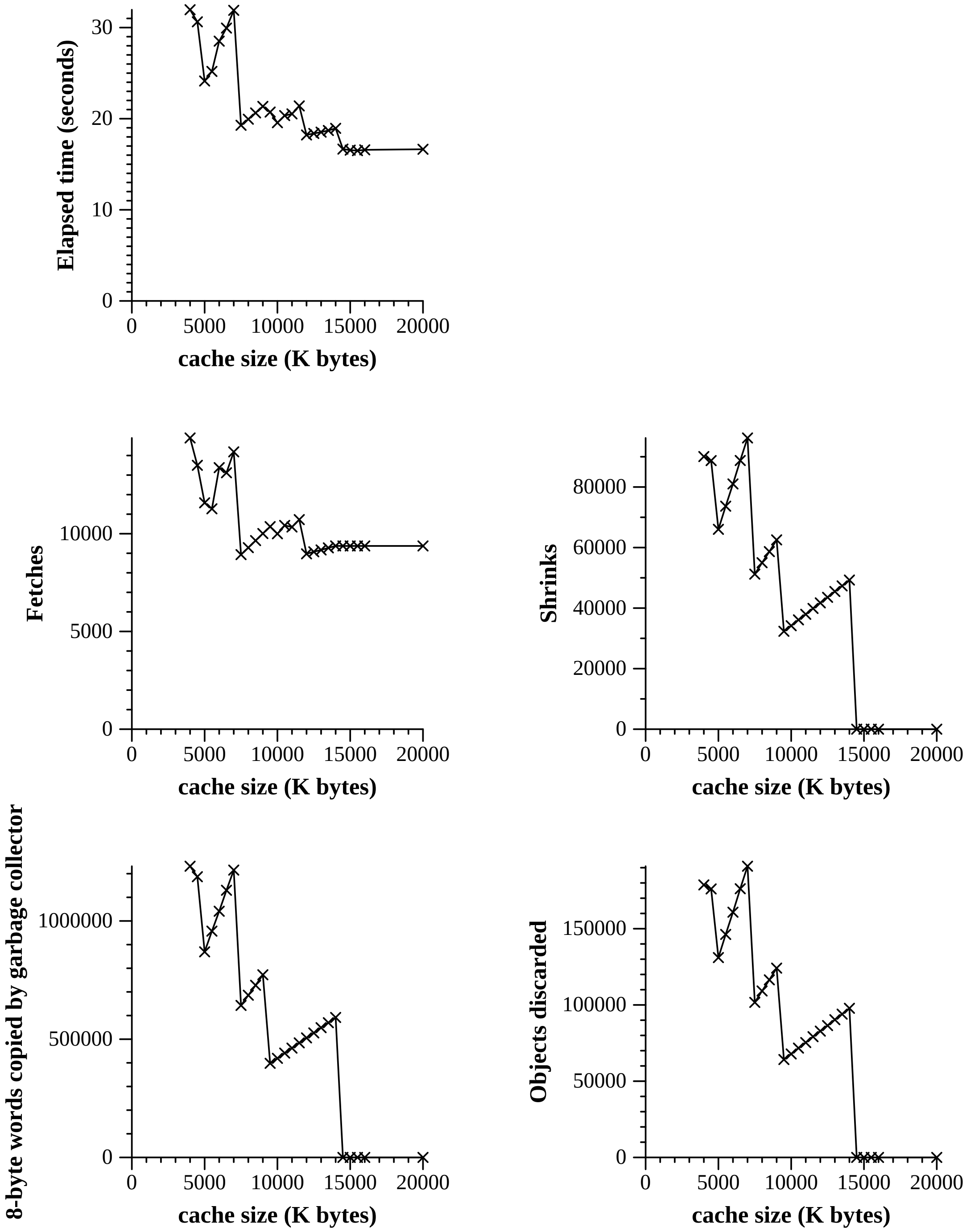

Figure 7-6: Example shrinking policy: TOTAL

graphs are presented roughly in order of significance. The top graph shows elapsed (wall-clock) time as measured by the on-chip cycle counter of the workstation. The two graphs immediately below the graph of elapsed time show the number of fetch requests sent to the server and the number of words copied by the garbage collector. Fetch requests are typically the dominant cost in the system, and garbage collection costs are secondary. The two graphs in the right hand column show different aspects of shrinking. These graphs have relatively little direct connection to the elapsed time, but they are useful for showing the behavior of the system. The upper right graph shows the number of objects chosen to be shrunk. The lower right graph shows the number of objects actually discarded from the cache. For this policy in this configuration, the graphs have the same shape; but in general, shrinking objects may make some other objects unreachable, so that the garbage collection after shrinking may reclaim more objects than just the ones shrunk.

In addition to measuring the number of objects reclaimed, I also measured the storage reclaimed (measured in bytes). Because the average object size is relatively constant in different parts of the overall traversal, the curve for the number of *bytes* reclaimed always had the same appearance as the curve for the number of *objects* reclaimed; therefore the bytes-reclaimed data is not presented here.

The shapes of the graphs are rather complex. Before considering variations in parameters or comparing policies, I describe some of the features of these graphs in more detail.

The shifting traversal does not create very much garbage: in the absence of shrinking, it essentially has a monotonically growing heap until it ends. Note that the largest cache size for which the garbage collector runs is actually somewhat larger than the entire database used. Since the garbage collector can only run between operations, it must run slightly before the cache is full; the gc trigger parameter controls how close to the edge the system runs before starting a collection. If the cache manager somehow knew the computation in advance, it could avoid performing a garbage collection at such a point; but in the absence of such knowledge, running out of cache is a much more serious performance problem than excessive garbage collection, since the former means that the system stops while the latter only slows it down.

Shrinking is triggered as a consequence of the garbage collector failing to free enough space. In this data, shrinking is triggered after all the live objects have been copied once, and (with the TOTAL policy) causes essentially all of those live objects to be shrunk. When there is little enough storage that garbage collection is needed, there is roughly a one-to-one correspondence between objects copied, objects shrunk, and objects discarded; so it is not surprising that the curves for these quantities have identical shapes.

Two features of these graphs are somewhat puzzling. First, the four identically-shaped curves show an odd "stair step" pattern. Second, the number of fetches actually increases with some increases in cache size.

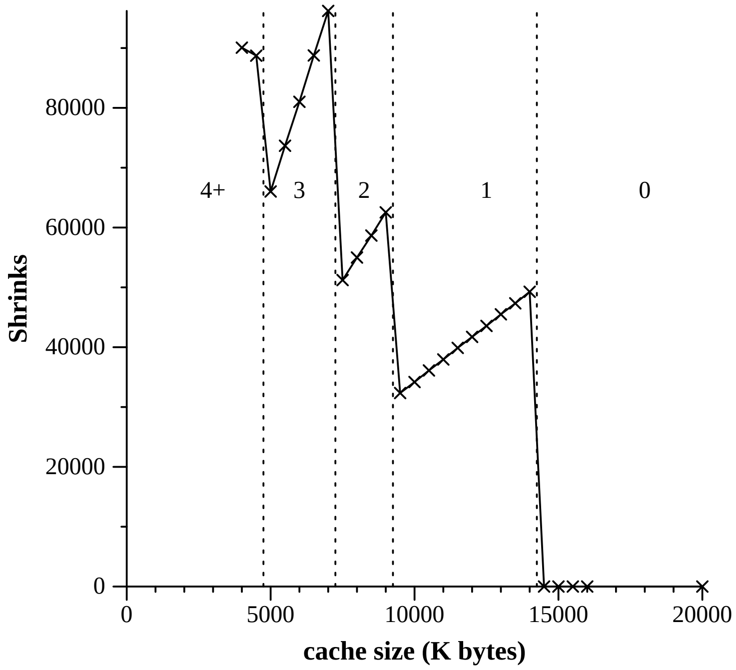

Figure 7-7: Number of times shrinking is triggered

The drops in the "stair step" pattern are simple to explain: each drop in the graph corresponds to a point at which the system does less shrinking than it did at the previous level. Figure 7-7 shows the number of objects shrunk and the number of times that shrinking is triggered. The flat portion of the graphs on the right represents cache sizes for which shrinking is unnecessary. The sharp drop at about 14000 Kbytes represents the transition to a cache large enough that shrinking is not triggered. Similarly, the sharp decrease below 10000 Kbytes comes about because the cache is large enough to trigger shrinking once instead of twice.

The slope between the jumps is caused by the changing cache size, since garbage collection is triggered by reaching a particular fraction of the cache size. As the cache size grows, a fixed fraction of the cache contains more objects to shrink when shrinking is triggered.

Returning to Figure 7-6: The variation in the number of objects fetched (middle left graph) is best understood as relatively small fluctuations due to the complex interaction between the prefetching mechanism and the point at which shrinking occurs. A later experiment (described in Section 7.5.3) varies the maximum size of the prefetch group. The results of that experiment show that the fluctuations in the lower left graph of Figure 7-6 are larger than the fluctuations observed for other prefetch group sizes, but the fluctuations are insignificant compared to the large overall differences in fetches for different prefetch group sizes.

The graphs in Figure 7-6 show that fetching is the dominant effect in determining the elapsed time: the curve for elapsed time (upper left) is quite similar in shape to the curve for fetches (lower left). The other factors in the system seem to have only a second-order effect. Intuitively, one would expect that fetching is the dominant effect, garbage collection is secondary, and shrinking costs are tertiary. The following approximate calculation supports that intuition: The fetching curve shows about 10^4 fetches occurring, and we know from Chapter 4 that each fetch costs on the order of 10^5 cycles. So we might expect the fetching to be taking 10^9 cycles. The copying curve shows 10^6 or more words copied for small caches. We might expect the garbage collector's cost (copying and various table updates) to work out to no more than 10-100 cycles per word. So the garbage collector could take no more than 10^8 cycles. The shrinking curve shows no more than 10^5 objects being shrunk. Even if the cost of each shrink were as high as 100–1000 cycles, the shrinking cost would be no more than 10^7–10^8 cycles.

7.5 How to Compare Shrinking Policies

Shrinking policies interact with other aspects of the system, such as prefetching and garbage collection. The following parameters seem likely to affect the behavior of the system when we compare shrinking policies:

- garbage collection trigger
- prefetch algorithm
- maximum prefetch group size
- shrink trigger
- shrink fraction

The garbage collection trigger determines when there is little enough free space left so that the garbage collector should run. It was described in Chapter 6. The prefetch algorithm determines which extra objects are sent from the server, and the prefetch group size determines how many extra objects may be sent from the server. Both were described in Chapter 5. We determined in Chapter 5 that bf-cutoff was a good prefetcher, and that a good prefetch group size for this database was about 30 objects. We further determined that this value was not affected by changing the size of the database as long as the objects themselves stayed roughly the same. Accordingly, we use those as the starting points for understanding how performance changes with varying parameters. The shrink trigger determines when the garbage collector has failed to recover enough storage, so that shrinking must occur. It was described in Chapter 6. The shrink fraction determines (for certain shrinking policies) what fraction of the cached shrinkable objects should be discarded from the cache when shrinking is triggered. It was also described in Chapter 6.

GC trigger	freespace $\leq$ 20000 8-byte words
Prefetching algorithm	bf-cutoff
Prefetch group size	$\leq$ 30
Shrinking trigger	freespace $\leq$ 50% of halfspace after gc
Fraction of objects shrunk	Not applicable

Figure 7-8: Starting values for parameters for TOTAL

These five parameters control shrinking and storage management, and it is not obvious how they affect system performance. By varying these parameters and comparing performance on the shifting traversal, it is possible to determine which parameters have the largest effect and to proceed accordingly.

7.5.1 Varying the GC trigger

Figure 7-9 shows the effect of varying the gc trigger point when using the TOTAL policy. In this and subsequent Figures, the default configuration's performance is shown as a solid line and all variations are dotted lines. Since most of the curves are quite similar, I show only the default gc trigger point (20000 8-byte words), a larger value (50000 words), and a smaller value (2000 words). A smaller gc trigger leads to fewer collections; a larger gc trigger leads to more collections. One would expect a smaller gc trigger to improve performance. However, a gc trigger point of 1000 words (not shown) is *too* small: the traversal fails to complete for some cache sizes.

The performance gain for a low gc trigger point is rather small, while there is an increased potential for the system to simply stop. Accordingly, it does not seem worthwhile to try for careful tuning of the gc triggering mechanism. Overall, variations in the gc trigger have little effect on the system's performance. There is no evidence to suggest that the gc trigger is critical to comparing shrinking policies.

7.5.2 Varying the Prefetch Algorithm

Figure 7-10 shows the effect of varying the prefetch algorithm when using the TOTAL policy. These graphs confirm what was demonstrated in Chapter 5: the simple bf-cutoff prefetcher is better than either depth-first or bf-continue prefetching, even though the bf-continue prefetcher fetches less often. In Chapter 5, there was no storage management, but we can see that the addition of storage management does not change the result. As in Chapter 5, the explanation is that each individual fetch with bf-continue is taking longer on average than is the case for bf-cutoff.

There are three other interesting features of these graphs. First, on the graphs dealing with storage management (bottom left and the two right graphs) the curves for bf-continue are shifted

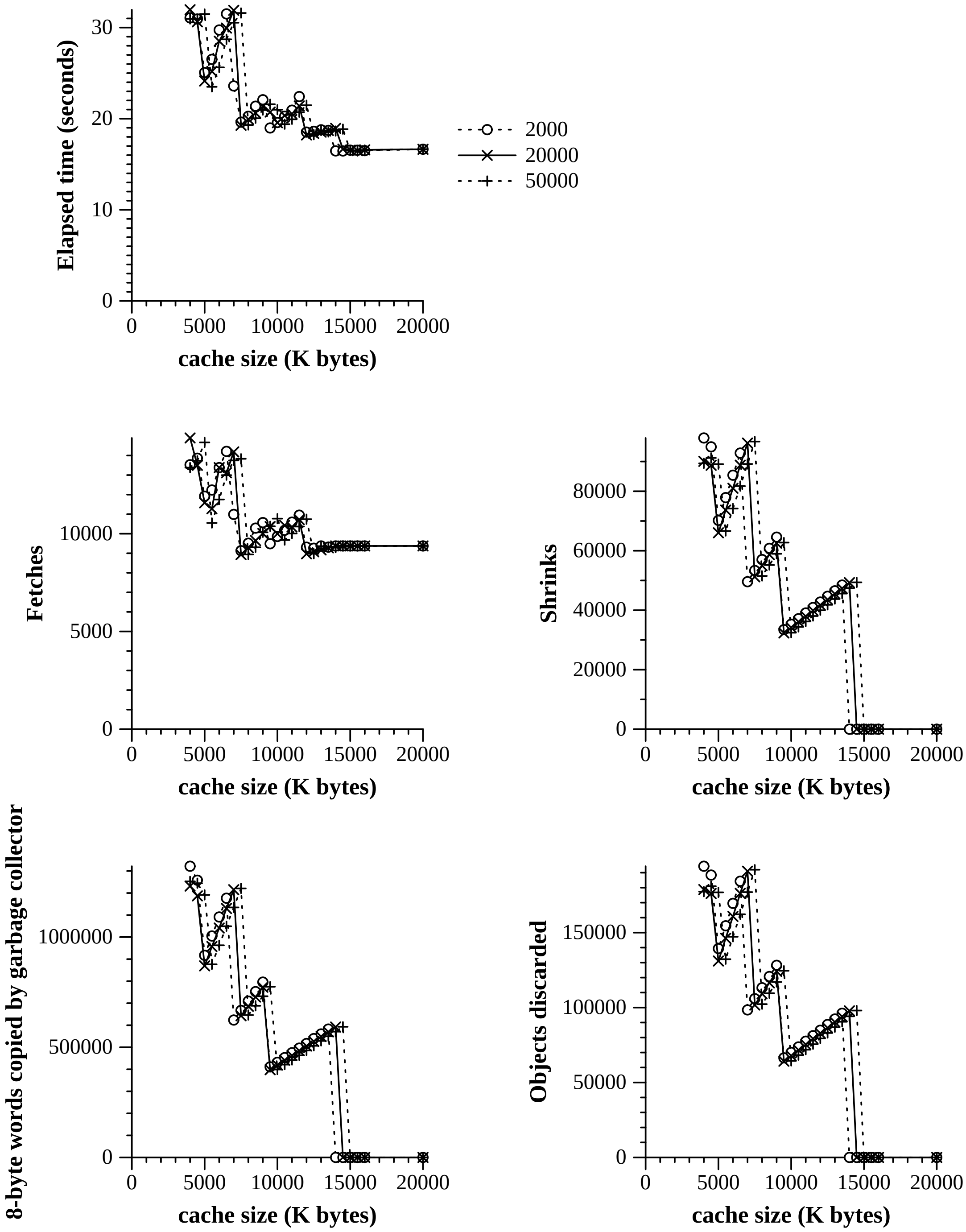

Figure 7-9: TOTAL, varying gc trigger

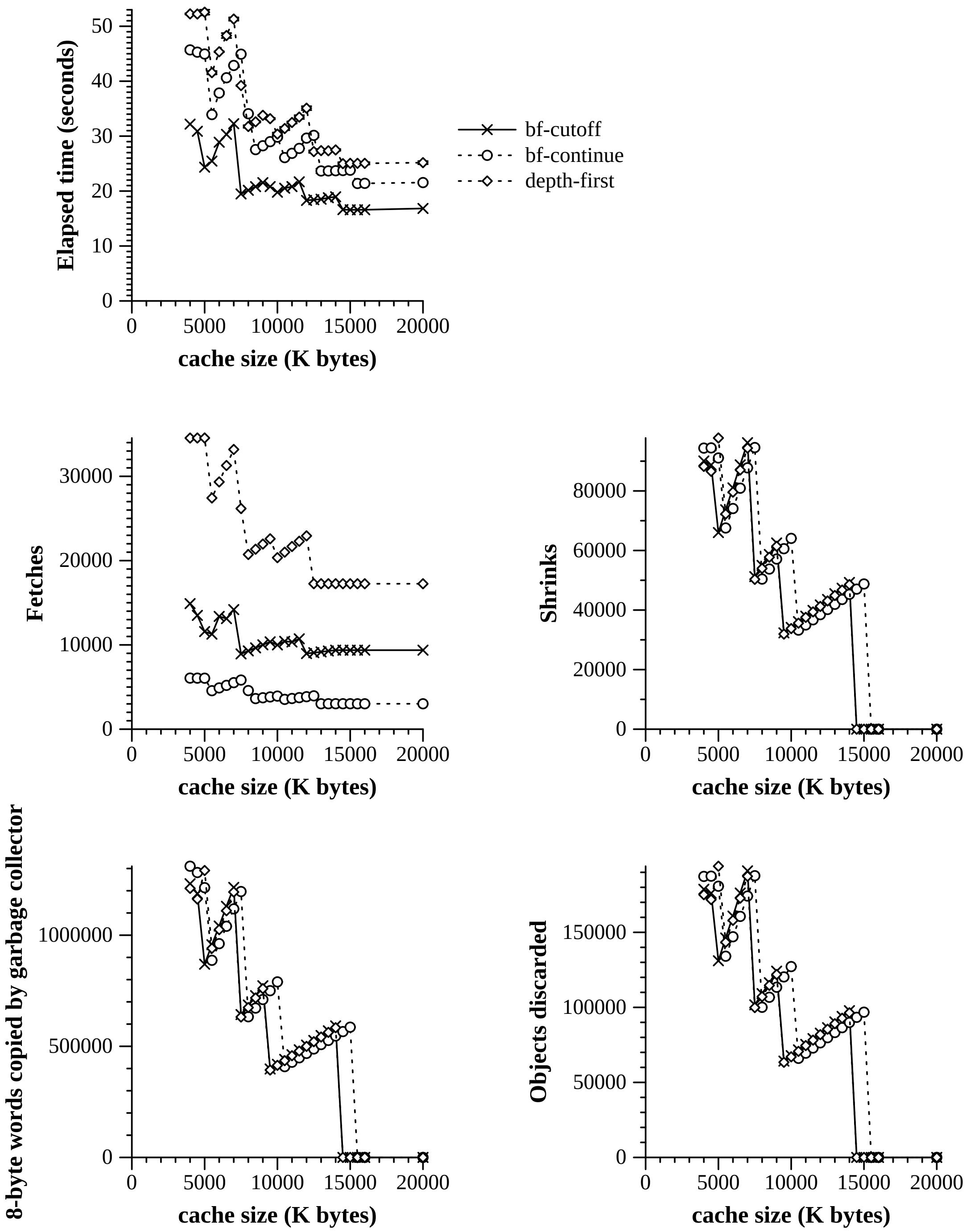

Figure 7-10: TOTAL, varying prefetch algorithm

to the right from the curves for bf-cutoff. The shift suggests that bf-continue is fetching 500–1000 Kbytes more than bf-cutoff or depth-first. Second, a comparison of the elapsed time curves (top) with the fetches curves (middle left) shows that bf-continue prefetching behaves quite differently from bf-cutoff or depth-first prefetching: the depth-first and bf-cutoff curves have approximately the same shape and relative position on the two graphs, whereas bf-continue changes both its shape and relative position. Third, the different policies show a large difference in the graph of fetches, but very little difference in the graphs related to storage management (words copied, shrinks, or objects discarded). The prefetch algorithm, like the gc trigger, seems to have little effect on storage management.

7.5.3 Varying the Maximum Prefetch Group Size

If the maximum prefetch group size gets smaller, we would expect system performance to degrade. Figure 7-11 shows the effect of decreasing the maximum prefetch group size when using the TOTAL policy.

In Figure 7-11, decreasing the maximum prefetch group size causes elapsed time (top) to increase dramatically, especially when prefetching is eliminated (a prefetch group of 1 means single-object fetching). The graph of fetches (middle left) shows that most of this effect is due to increased fetching, as might be expected.

Figure 7-11 also shows that decreasing the size of the maximum prefetch group has a benefit: it decreases the cache size at which storage management is needed. A single-object fetching system can complete the shifting traversal in a 10000 Kbyte cache without needing to do any garbage collection or shrinking at all; the default configuration requires more than 15000 Kbytes of cache to avoid using garbage collection or shrinking. However, this effect is not critical to performance. Despite the overhead of storage management, the default configuration (with 30-object prefetch group) is more than twice as fast for a 10000 Kbyte cache as the single-object fetching system.

Figure 7-12 shows the effect of *increasing* the maximum prefetch group size when using the TOTAL policy. In contrast to the large effect shown in Figure 7-11 for decreasing the prefetch group size, Figure 7-12 shows that increasing the maximum prefetch group size has little effect on elapsed time (top graph). We already know from Chapter 5 that increasing the maximum prefetch group size hurts the performance of the system on sparse traversals. The potential gain with higher prefetch levels is much smaller than the potential losses with lower prefetch group sizes.

The number of fetches fluctuates sharply for a maximum prefetch group size of 75. Recall from Chapter 5 that even on a dense traversal, a higher prefetch level sometimes (paradoxically) causes more fetching. When comparing the elapsed time graphs in Figures 7-11 and 7-12, the reader should note the difference in the scales used on the respective y axes. The fluctuations in elapsed time that seem quite large in Figure 7-12 are actually small compared to the differences shown in Figure 7-11.

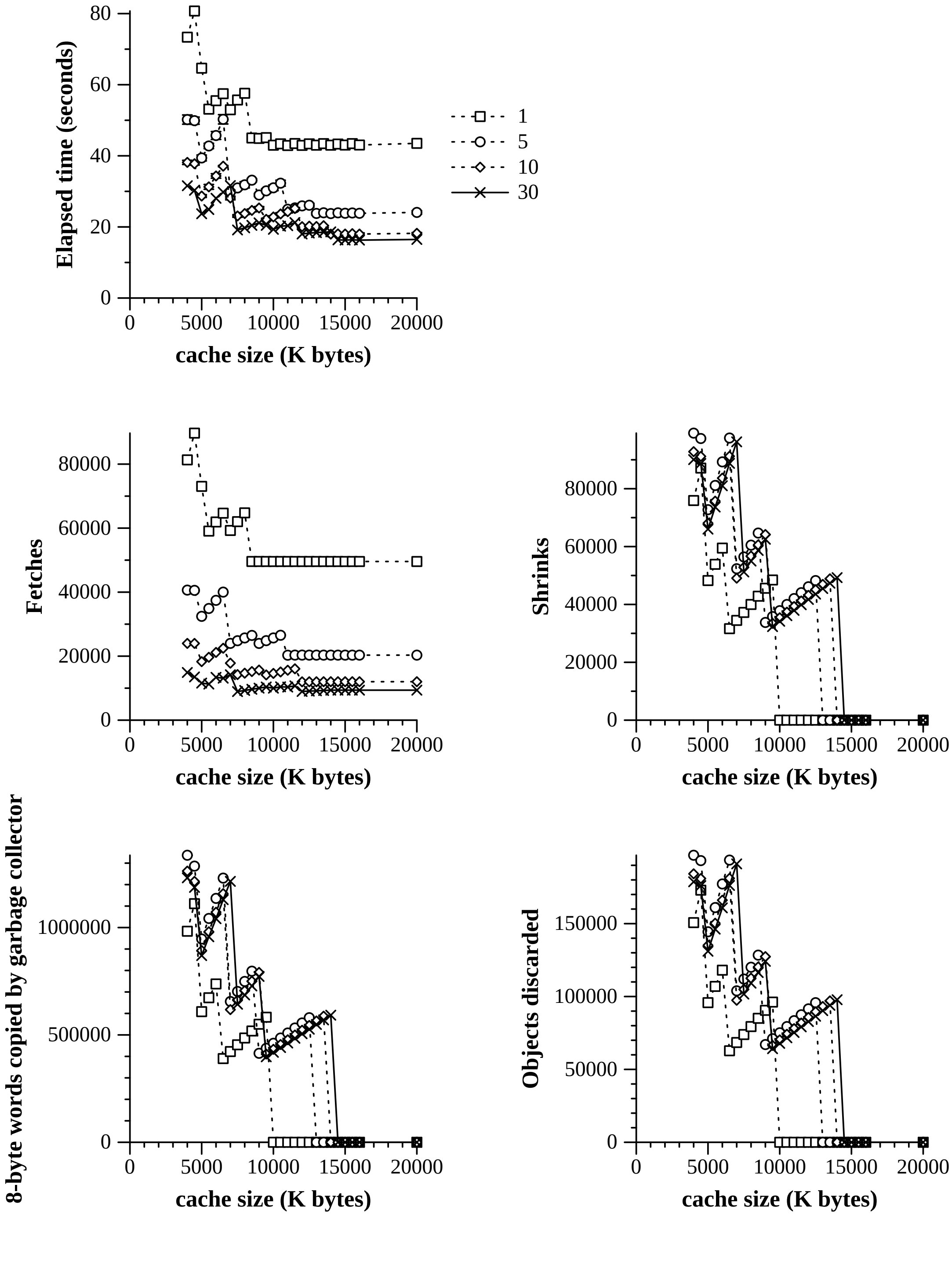

Figure 7-11: TOTAL, decreasing maximum prefetch group size

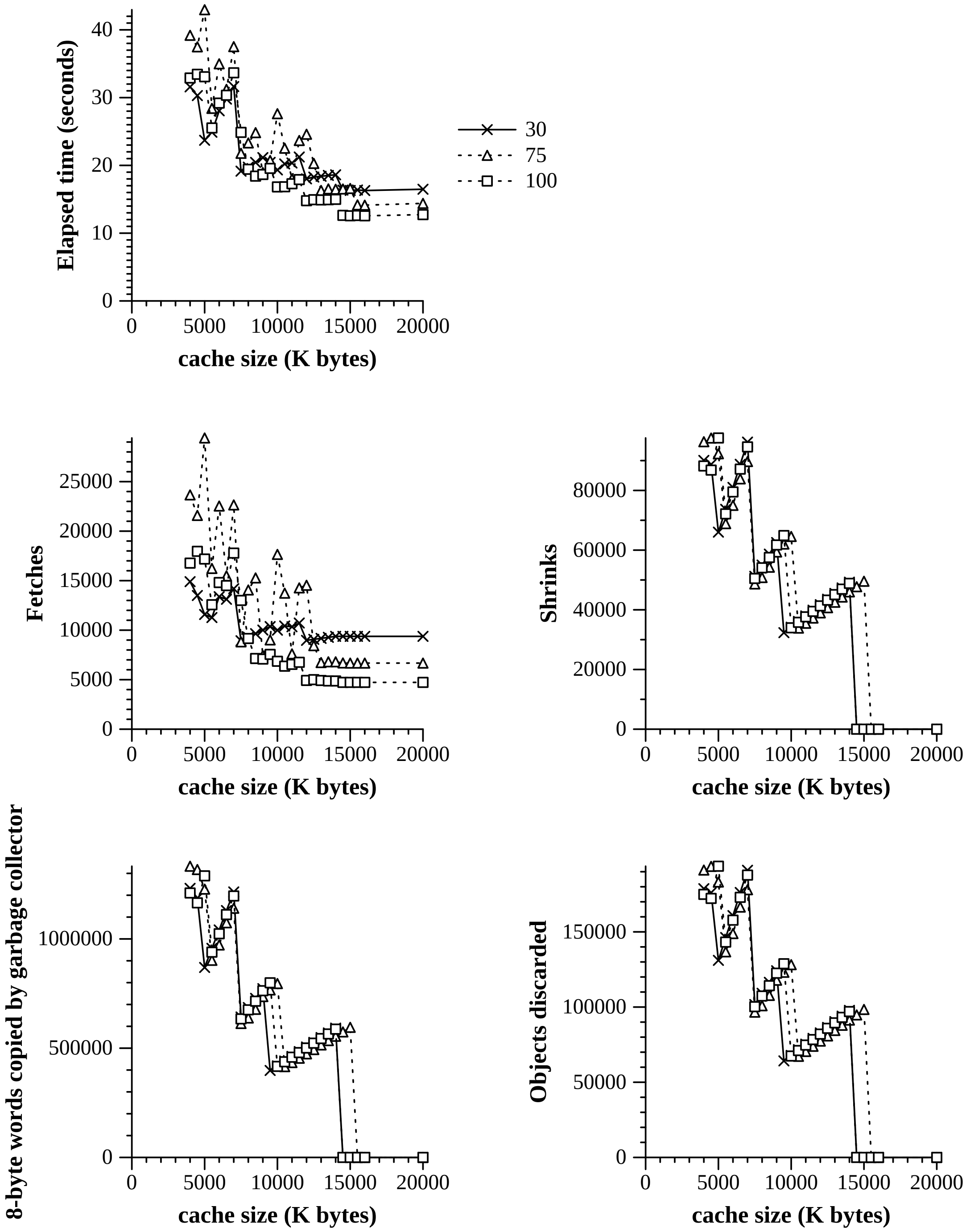

Figure 7-12: TOTAL, increasing maximum prefetch group size

7.5.4 Varying the Shrink Trigger

The shrink trigger controls the point at which the system decides to shrink. A low shrink trigger means that the system shrinks objects more readily. The default shrink trigger is set at 0.5, meaning that shrinking is triggered whenever a garbage collection fails to recover at least 0.5 of the heap (since it is a copying collector, 0.5 of the heap means 0.5 of a halfspace, or 0.25 of the whole cache). Figure 7-13 shows the effect of varying the shrink trigger when using the TOTAL policy. A shrink trigger of 0.05 or less has much worse performance for smaller caches than other shrink triggers. The graph of gc copying (lower left) shows that for a shrink trigger of 0.05 there is an explosion of garbage collection, repeatedly copying objects that should have been shrunk.

There are two broad regions in which the system's performance does not change with small changes in the shrink trigger: one is with the shrink trigger in the range 0.2 to 0.35, and the other is with the shrink trigger in the range 0.45 to 0.9. Since the default value of 0.5 is in the latter range, that range is plotted with a solid line.

Figure 7-14 shows shrink triggers smaller than the lower region. Shrink triggers smaller than 0.2 show excessive garbage collection (middle left), similar to what was apparent for a shrink trigger of 0.05, although not as extreme. A shrink trigger in the range 0.2 to 0.35 shows the best performance in terms of fetches (lower left) and elapsed time (upper left). Higher values for the shrink trigger cause more shrinking to happen, causing less garbage collection but more fetching. Since refetching an object is more expensive than copying it during garbage collection, the higher shrink triggers show higher elapsed times.

Being too reluctant to shrink causes high costs due to excessive garbage collection: live but irrelevant data is being copied and recopied, when it would be better to discard it and refetch it if needed. The excessive cost is not an artifact of using a copying collector: a mark/sweep collector would be similarly penalized for keeping too much live but irrelevant data, since garbage collections would be triggered more frequently and both marking and sweeping would take longer.

7.5.5 Better configuration for TOTAL

Figure 7-15 summarizes a better configuration for the TOTAL policy, based on the experiments of this section. The only significant change is that shrinking is triggered when 30% or less of halfspace is free after a garbage collection, whereas the initial configuration used 50% as the trigger. Figure 7-16 compares this better configuration to the original configuration.

7.5.6 Discussion

This section has presented a detailed examination of all the parameter variations for TOTAL. A similar collection of experiments was performed for the other policies, confirming the overall con-

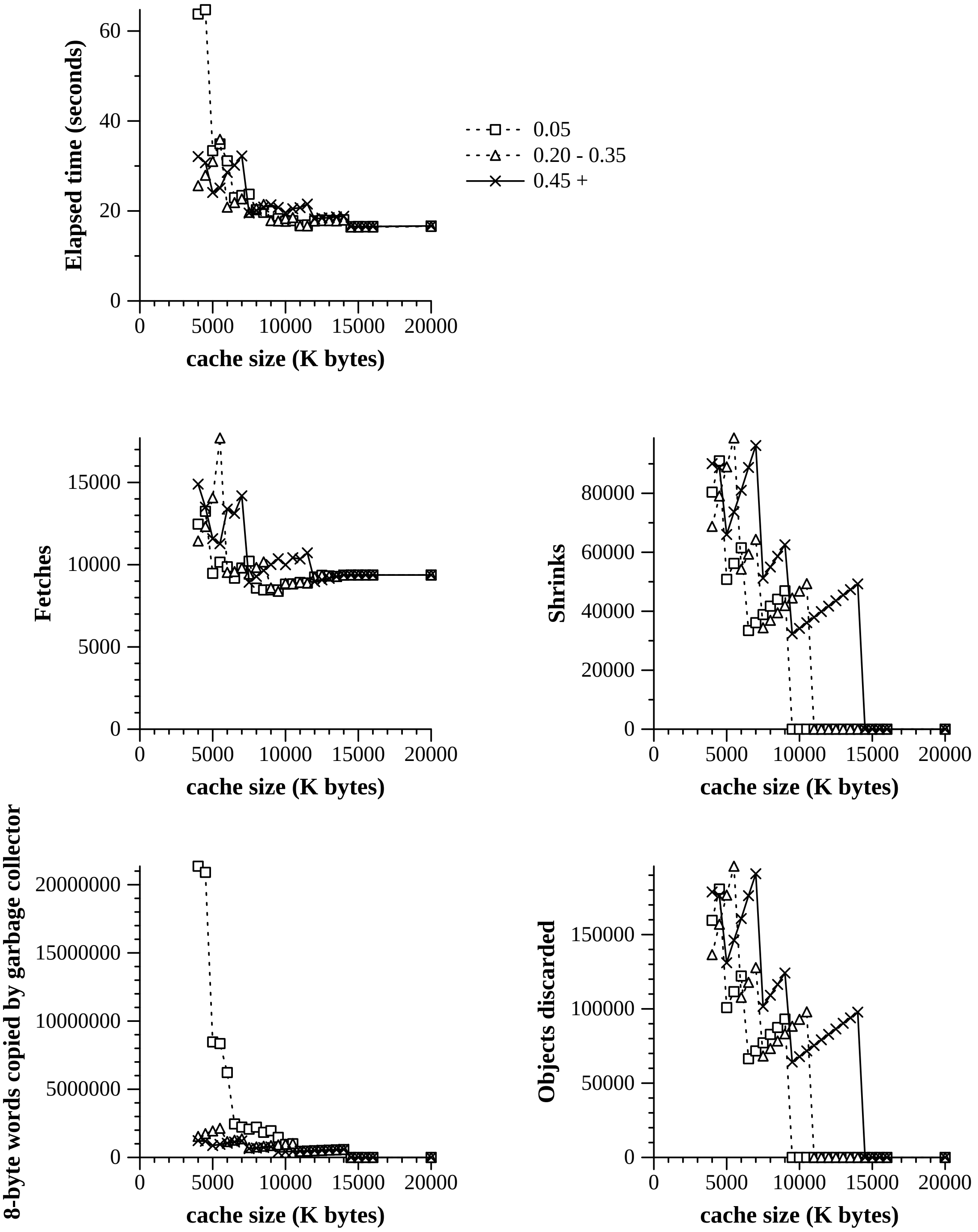

Figure 7-13: TOTAL, varying shrink trigger

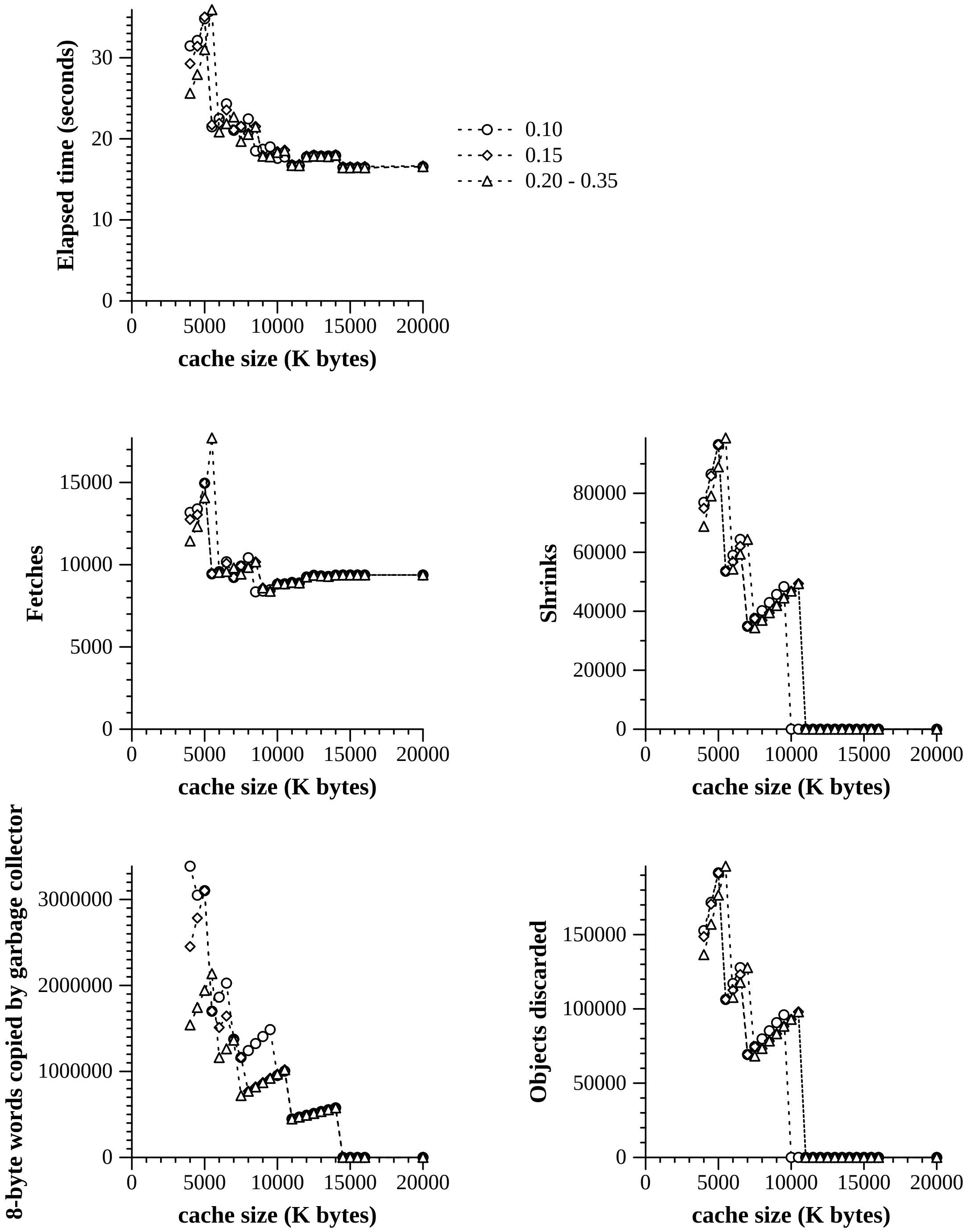

Figure 7-14: TOTAL, very small shrink triggers

Prefetching algorithm	bf-cutoff
Prefetch group size	≤ 30
GC trigger	freespace $\leq$ 20000 8-byte words
Shrinking trigger	freespace $\leq$ 30% of halfspace after gc *(changed)*

Figure 7-15: Better values for TOTAL parameters

clusions found here. The graphs for those experiments appear in Appendix B. To summarize that data, three of these parameters — gc trigger, prefetching algorithm, and prefetch group size — are independent of the shrinking policy chosen, and experiments showed that they have very similar effects on performance with any of the shrinking policies. The shrink trigger and shrink fraction affect the different shrinking policies somewhat differently, but neither parameter has a very large effect on overall performance of any policy. A reasonable rule of thumb is that the choice of prefetch algorithm and prefetch group size dominate any other parameter choice for any single shrinking policy.

7.5.7 TOTAL vs. CLEARCACHE

It is worth distinguishing TOTAL from another common policy with which it might be confused. In this policy (which I call CLEARCACHE), the system clears the cache at the end of each transaction. This cache-clearing takes place only because of the transaction boundary, which in turn is simply due to the consistency requirements of the computation, not because of any actual storage management need at that point in time. The CLEARCACHE policy has the advantage that it simplifies some aspects of cache management. However, it can have performance problems compared to a policy like TOTAL that clears the cache only when the cache fills.

In Figure 7-17, the traversal is treated as a series of smaller atomic transactions. Each elementary traversal making up the traversal is considered to be a distinct transaction. These may be thought of as atomic transactions, but in the experiment the commit at the end of each transaction takes no time at all. For these read-only transactions at a single site, a commit would actually require about the same amount of time as a fetch; compared to the number of fetches in each transaction, this overhead is indeed quite small. However, if the transactions involved modifications or multiple sites, the cost of each commit would be comparable to several fetches.

CLEARCACHE works well if successive transactions use disjoint sets of objects. However, there are two ways that a policy like CLEARCACHE can perform poorly. First, when successive transactions have a number of shared objects, as in this example, a policy like CLEARCACHE discards a number of objects at the end of a transaction that must be refetched by the following transaction. Figure 7-17 shows that the system performs much better by using a policy (such as TOTAL) that only clears the

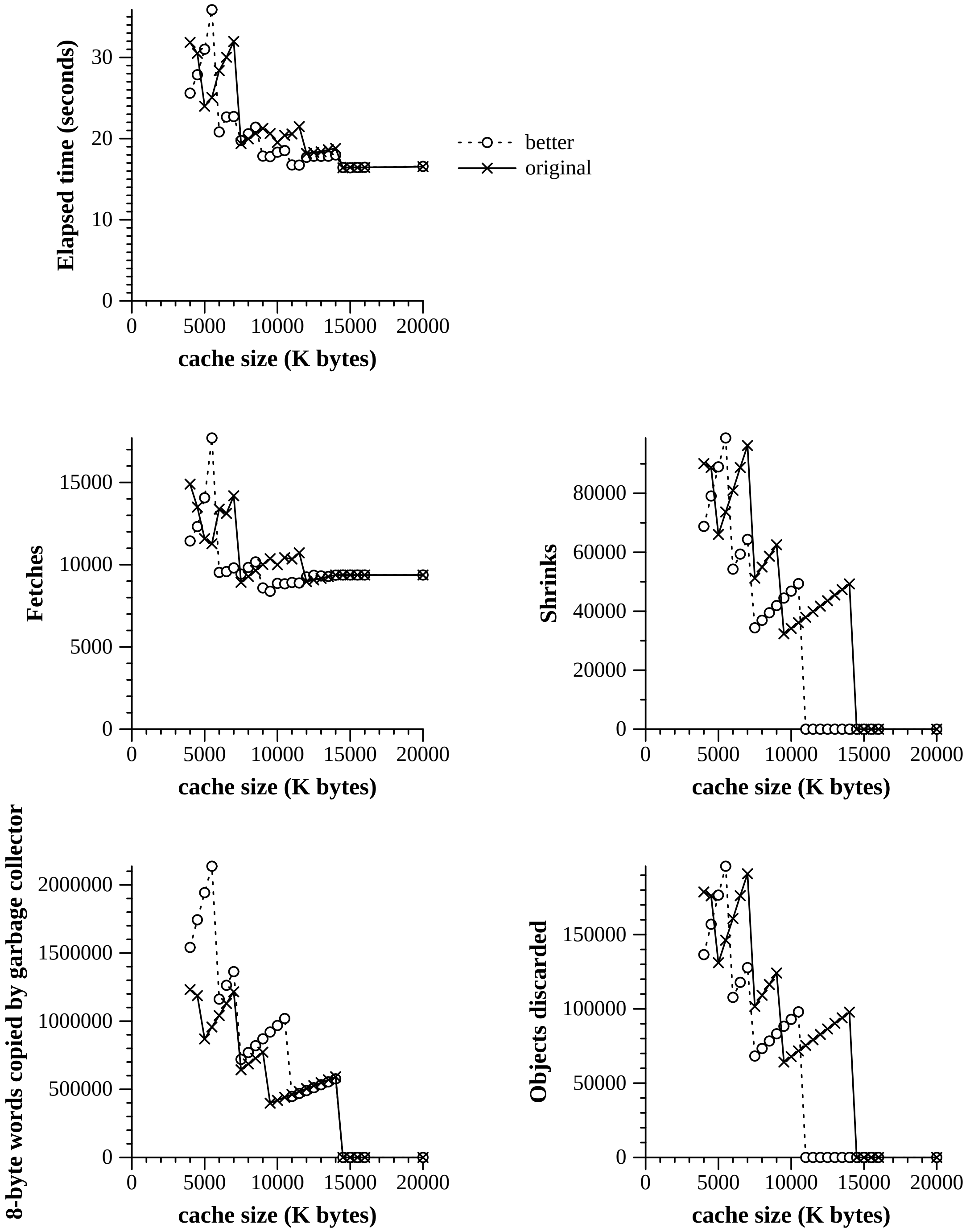

Figure 7-16: TOTAL, better vs. original configuration

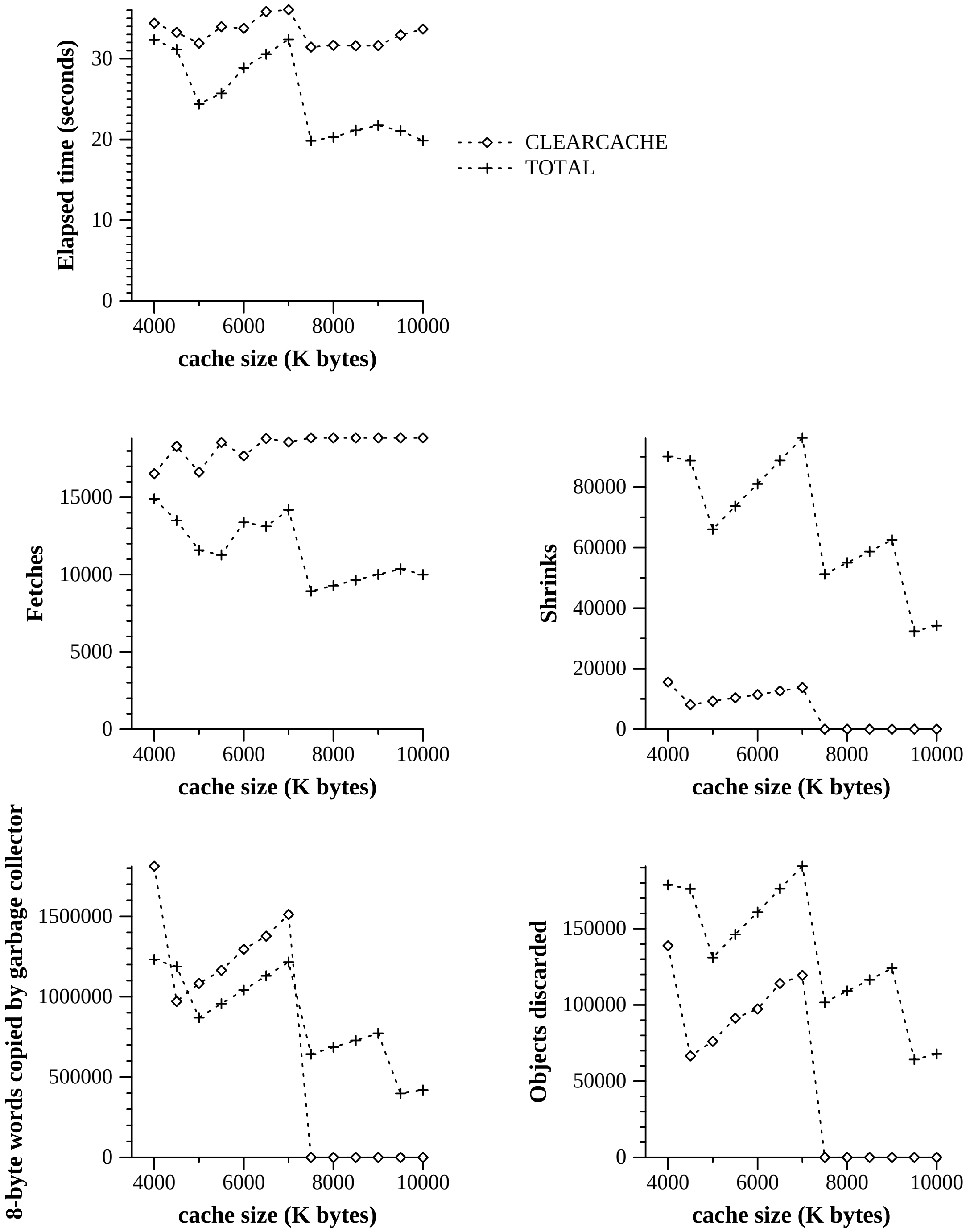

Figure 7-17: CLEARCACHE vs. TOTAL

client cache when it has filled up. The second way that CLEARCACHE can perform poorly is when a transaction reads more objects than can fit into the client's cache. If the cache can be emptied only at a transaction boundary, a transaction can fill the cache and then be unable to proceed. In contrast, a policy like TOTAL allows a larger number of objects to be fetched, read, and then shrunk as needed.

Until the cache gets very small, the CLEARCACHE policy performs worse than TOTAL. This is a substantial argument against CLEARCACHE when subsequent transactions have related data objects, especially since TOTAL does not do particularly well compared to other more sophisticate policies examined later in this chapter.

7.6 Comparing Shrinking Policies

Figure 7-18 shows MINFETCH, an unachievably good policy, and MAXFETCH, its unachievably bad mirror image. Between the two, but much closer to MINFETCH, lies the region of policies tested in this chapter. In this Figure, the realistic policies are plotted using the same mark so as to define a region of curves rather than specific curves. Note that the MAXFETCH curve begins at a cache size of 9500 Kbytes; for the configuration used, it was not possible to complete the benchmark traversal using a MAXFETCH policy and smaller caches. The graph of fetches shows that the MINFETCH policy had the lowest number of fetches, as expected.

7.6.1 TOTAL

Figure 7-20 compares TOTAL (which we have already considered in some detail) with MINFETCH. Figure 7-19 shows the parameters used for TOTAL in this comparison. Both TOTAL and MINFETCH are in configurations chosen to minimize the area under the curve of elapsed time. Accordingly, MINFETCH has a lower elapsed time for most cache sizes, but not for all. As we have noted previously, the performance of TOTAL is relatively erratic: there are large fluctuations in performance for small changes in cache size. MINFETCH is more consistent. For MINFETCH, the number of fetches actually increases with increasing cache size. This apparent anomaly simply reflects the fact that MINFETCH is better than the bf-cutoff prefetcher at determining which objects should be present in the cache. When the cache is smaller, MINFETCH is called on more often. It is not surprising that MINFETCH should do better than the prefetcher, since MINFETCH actually has access to the entire future computation.

7.6.2 RANDOM

With the RANDOM shrinking policy, a coin is tossed for each shrinkable object. The shrink fraction determines the bias on the coin, and accordingly the fraction of object shrunk is only probabilistically

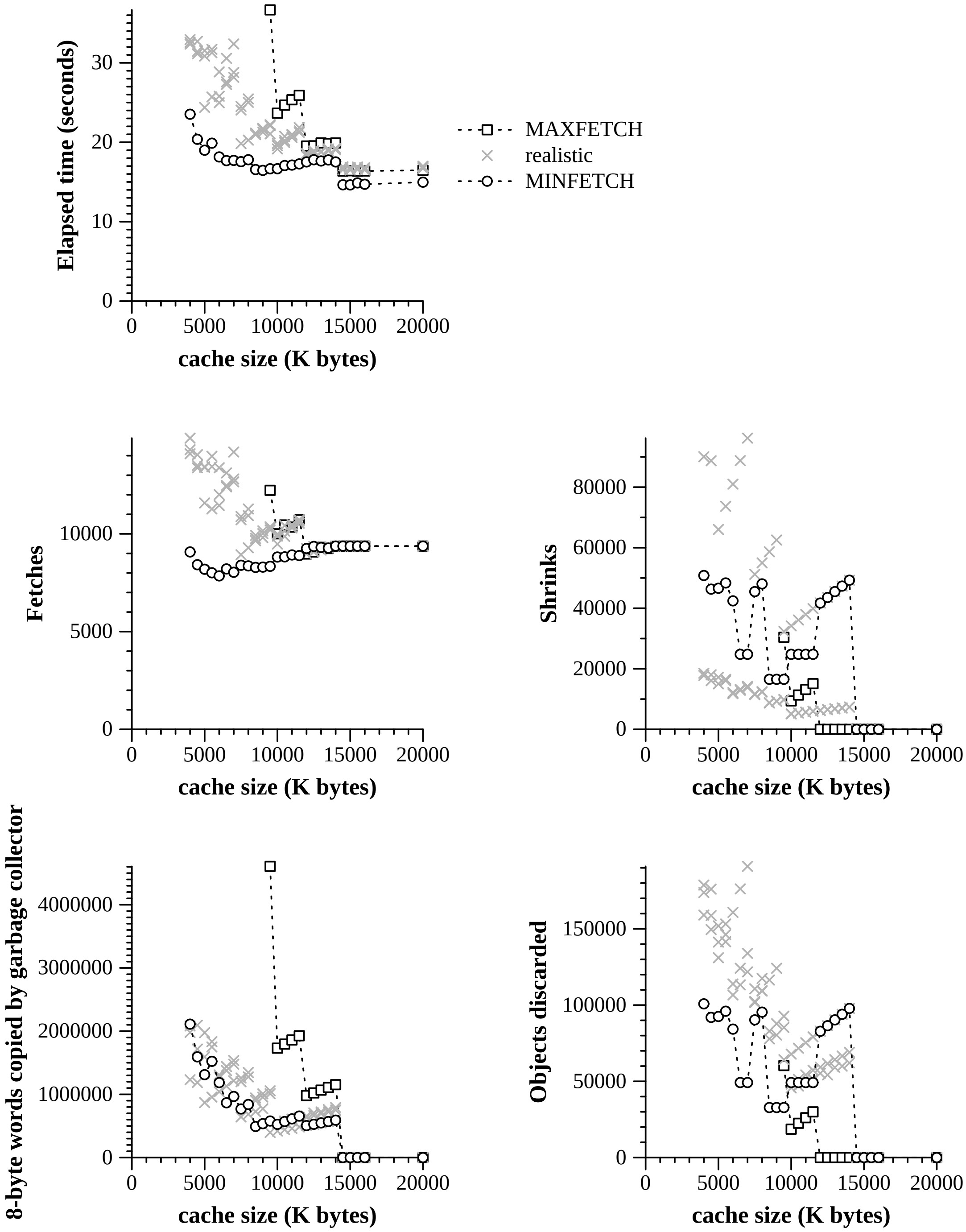

Figure 7-18: Best, worst, and realistic shrinking policies

Shrinking trigger	freespace $\leq$ 30% of halfspace after gc
Fraction of objects shrunk	Not applicable

Figure 7-19: Good parameters for TOTAL

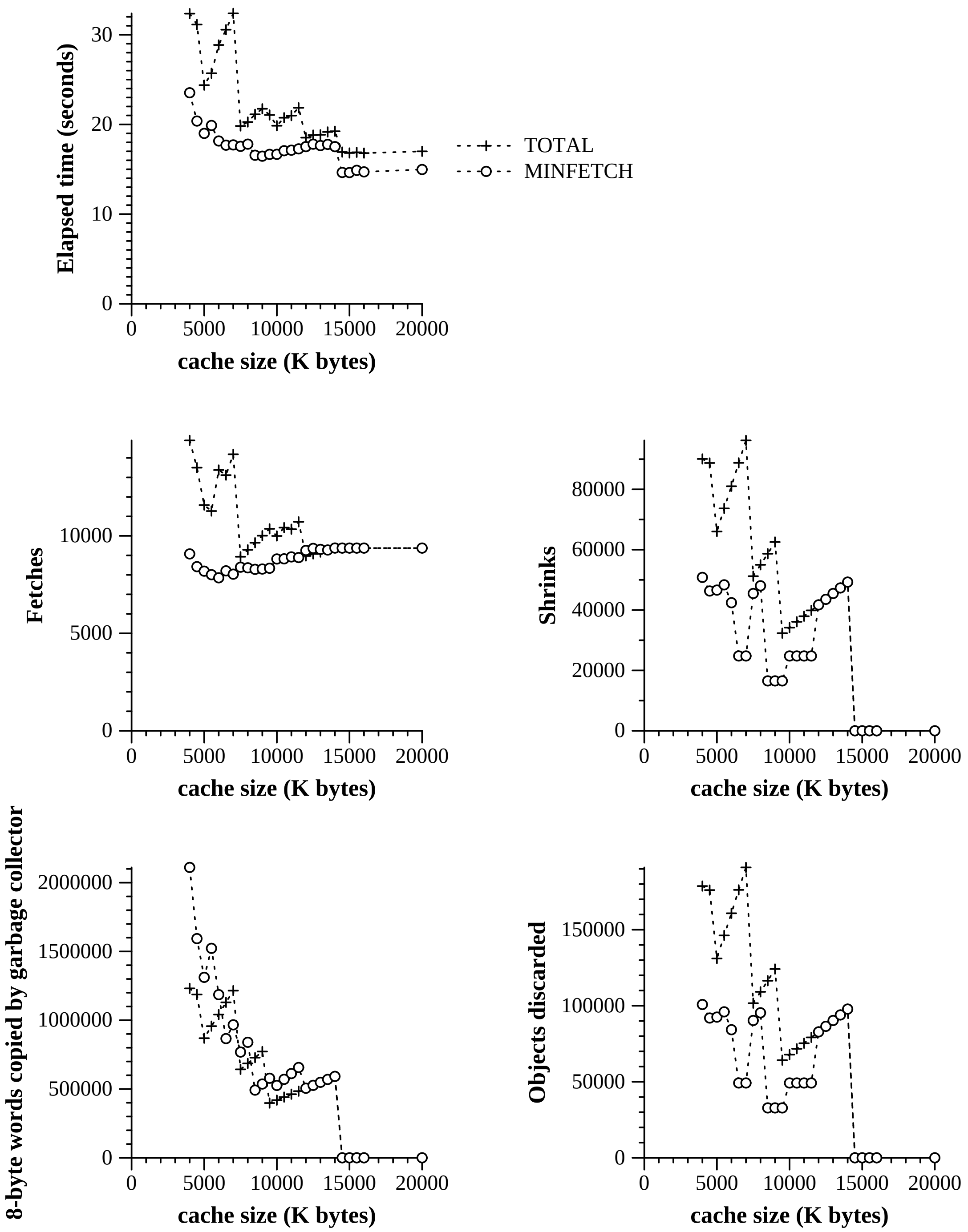

Figure 7-20: TOTAL vs. MINFETCH

Shrinking trigger	freespace $\leq$ 30% of halfspace after gc
Fraction of objects shrunk	15%

Figure 7-21: Good parameters for RANDOM

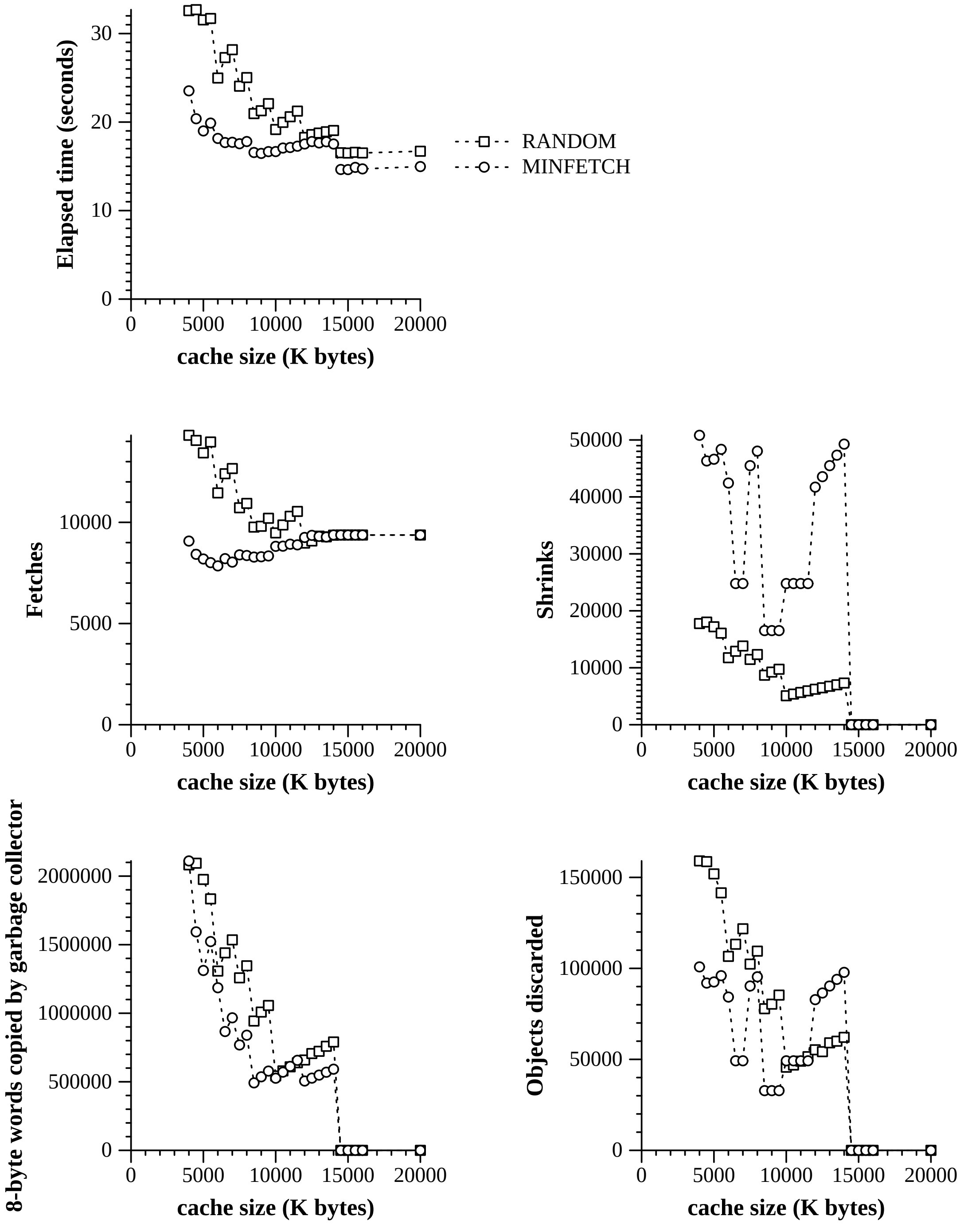

Figure 7-22: RANDOM vs. MINFETCH

Shrinking trigger	freespace $\leq$ 30% of halfspace after gc
Fraction of objects shrunk	15%

Figure 7-23: Good parameters for CLOCK

close to the shrink fraction specified. However, for the benchmarks used, this mechanism worked well and produced shrinking comparable to what was achieved by other, more careful mechanisms, as can be seen by comparing the curves for objects discarded for the RANDOM and CLOCK policies (CLOCK appears in the next section). Figure 7-22 compares RANDOM with MINFETCH. The curve of shrinks (upper right) for the RANDOM policy is quite different from the curve of objects discarded (lower right), in contrast to what is shown for TOTAL in Figure 7-20. This suggests that with a RANDOM selection of objects to be shrunk, the relationship between objects chosen and storage recovered is more complex than is true for TOTAL, which matches intuition. Note also that RANDOM shrinks many fewer objects than TOTAL, since it is shrinking only 15% of the objects instead of 100%.

7.6.3 CLOCK

With the CLOCK pseudo-LRU shrinking policy, each object has a "use" bit that is set when the object is used by the client computation. The system chooses an object to shrink by proceeding sequentially through the table of objects present at the client, starting each time at the entry where it ended the previous time. If the object's "use" bit is set, that bit is cleared. If that bit is clear, the object is shrunk. The algorithm terminates when enough objects have been shrunk to meet the shrink fraction, or after cycling through the entire table of objects present at the client.

The overall performance of CLOCK is very similar to that of RANDOM. Apparently the single bit per object is not retaining enough information about access patterns, so that the effect is no different from a random distribution of bits. In contrast to both RANDOM and TOTAL, there is a runtime cost for CLOCK even when no shrinking is needed: the system must keep setting the "use" bits.

At the beginning of any method that reads or writes a field of a mutable object, a prologue is executed that performs the housekeeping necessary to track which objects have been read by the current transaction. After the first time that an object has been read, subsequent invocations of that method incur a cost of only $4 + 2J$ cycles. Modifying the prologue to support the setting of a use bit in the object requires two additional instructions, for a cost of $6 + 2J$ cycles. Using the value $J = 12$ determined earlier, we can determine that the CLOCK scheme requires 30 cycles compared to 28 cycles with no overhead, or about a 7% *maximum* increase. Note that for a method that is longer, the two-instruction overhead added by CLOCK is proportionally smaller. For the benchmark used, there was no measurable sign that maintaining the use bits for CLOCK was slowing down the system.

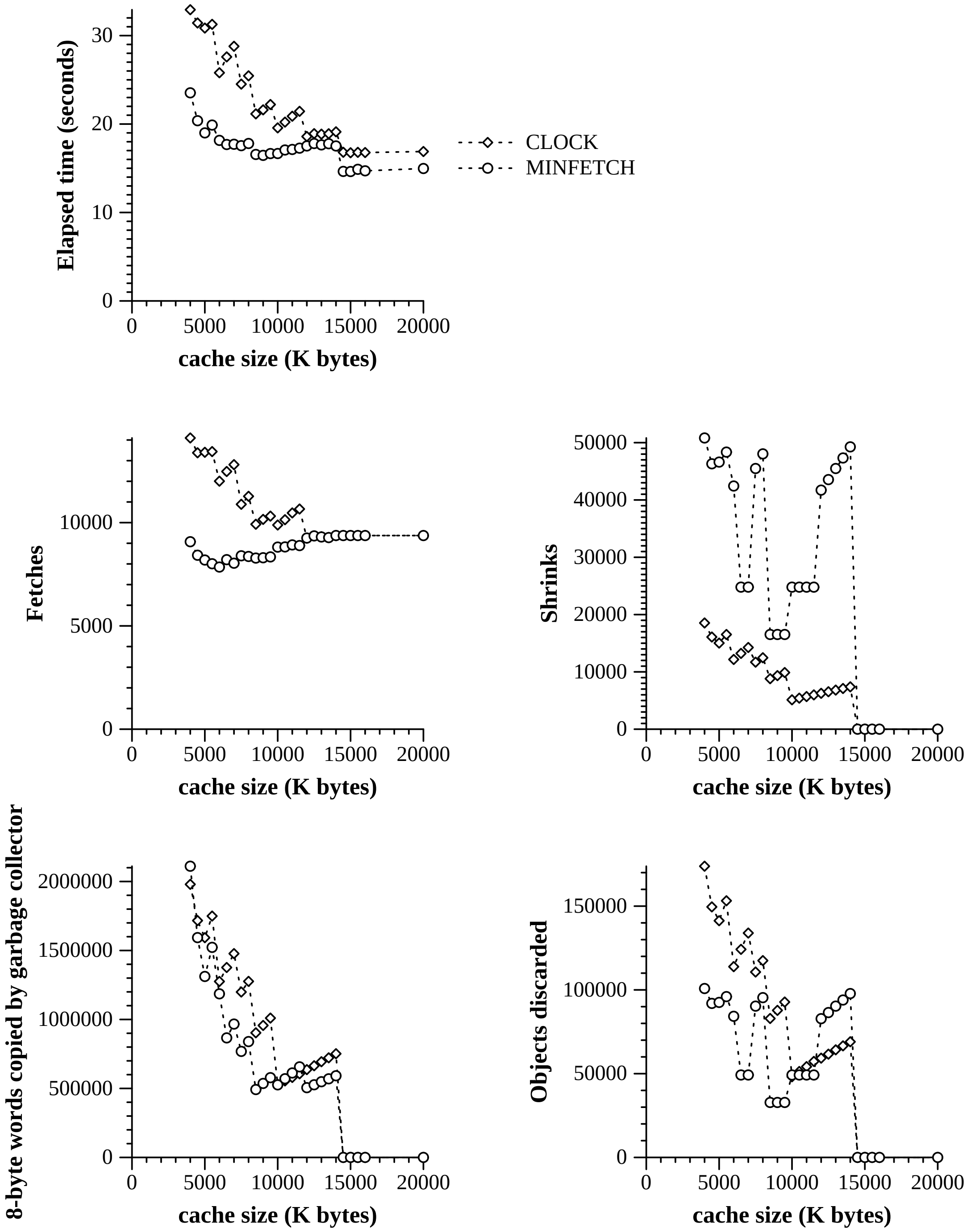

Figure 7-24: CLOCK vs. MINFETCH

Shrinking trigger	freespace $\leq$ 30% of halfspace after gc
Fraction of objects shrunk	50%

Figure 7-25: Good parameters for LRU

7.6.4 LRU

Whereas the CLOCK policy uses a single bit per object and sets or clears that bit, the LRU policy uses a 31-bit timestamp on each object. Each time an object is used, the timestamp is incremented and stored in the object used. The policy has a parameter that estimates the frequency with which each object is touched by the computation. The difference between the current timestamp and the timestamp at the last shrink provides the total number of references since the last shrink. By using the total number of references and the estimate of multiple references to the same object, the policy estimates a threshold timestamp that should shrink the desired shrink fraction. All of the objects below that threshold are shrunk. In the benchmarks used, the policy assumes that each object is used, on average, 3 times. This parameter behaves like many of the other shrinking-related parameters: for a fairly broad range of values, it affects the shrinking behavior observably but has relatively less effect on the overall elapsed time. For the value of 3, the shrink fraction specified is quite close to the fraction shrunk, so that a shrink fraction of 0.5 actually shrinks roughly 50% of the objects in the cache. The implementation of the LRU policy depends on modifying the method prologue in much the same way as was described for CLOCK. Instead of a 2-cycle overhead, LRU imposes a 5-cycle overhead, for a total cost of $9 + 2J$; however, as with CLOCK, this overhead is not large enough to be measurable with this benchmark.

The performance of LRU is very similar to the unattainable performance of the MINFETCH policy: both a small number of fetches and a low elapsed time. This suggests that a good, cheap, accurate LRU approximation can give very good shrinking performance. However, it is challenging to implement such an approximation for general workloads. The hard part of getting the LRU policy to work in practice is estimating the threshold timestamp for shrinking. Computing that threshold depends on estimating the average number of uses per object, and that estimate is impossible to provide *a priori* for an arbitrary workload.

A comparison of LRU and CLOCK is particularly interesting. The approximation of least-recently-used implemented by the CLOCK policy does not require any of the estimation needed for LRU, and is straightforward to implement for arbitrary computations. However, CLOCK has little advantage over mechanisms such as TOTAL and RANDOM that do not track usage information at all.

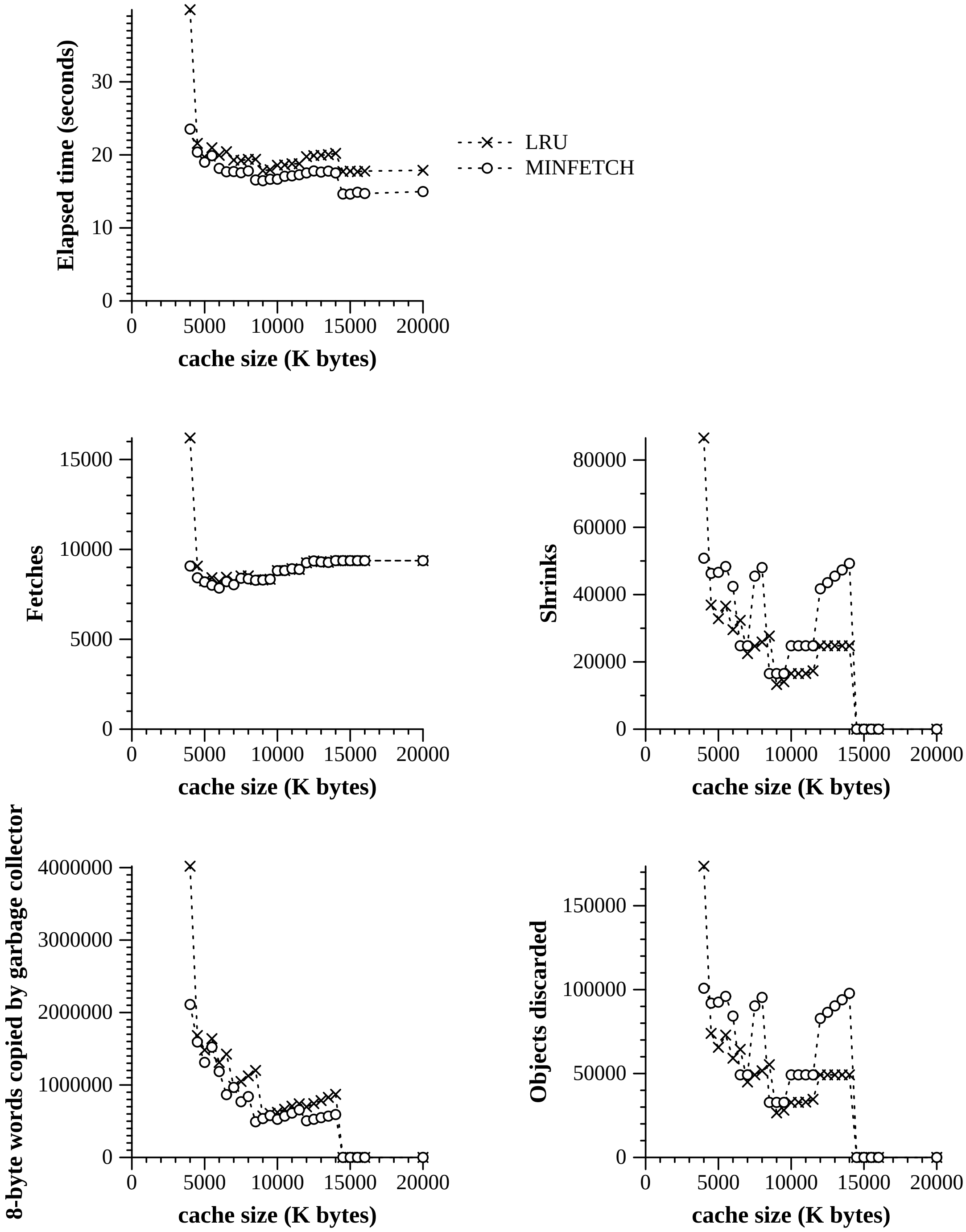

Figure 7-26: LRU vs. MINFETCH

7.6.5 Comparison

The significant differences in elapsed time occur only in the region between 4000 and 10000 Kbytes, which is approximately the region in which storage management is taking place but thrashing has not yet set in. That is the region that was shown in Figure 7-17, and is also the region shown in Figure 7-27. Figure 7-27 shows that LRU does consistently better than the other techniques. Among the other techniques, there is little meaningful difference between CLOCK and RANDOM. The TOTAL policy sometimes does better than CLOCK or RANDOM, and sometimes does worse; overall, TOTAL is more volatile than CLOCK or RANDOM. The TOTAL policy also shrinks the most objects, and causes the fewest objects to be copied by the garbage collector, both as expected. The MINFETCH policy causes the fewest objects to fetched, again as expected.

Since prefetching is so important to performance, it is possible that prefetching is masking some important difference among policies. Figure 7-28, which shows the effect of turning off prefetching, also shows that this possibility is not true. The largest effect shown by the graphs is that no matter what the cache management policy, the system is running a good deal slower: the best time is more than 40 seconds, compared with slightly less than 20 seconds with prefetching. The other, subtler effect is that TOTAL is now a fairly consistently bad policy. With prefetching, one might have argued for the value of TOTAL in some regions; if one knew that one's application were in a similar region, one could say that TOTAL was not only cheap to implement but offered reasonably good performance. In the absence of prefetching, it is apparent that TOTAL is basically a poor cache management policy at all cache sizes. The TOTAL policy, unlike the others, is made a good deal better by prefetching. The TOTAL policy has the disadvantage that it throws out potentially-useful objects, but it has the advantage that no space is consumed by useless objects; prefetching helps to compensate for throwing out too many objects.

7.6.6 Discussion

Garbage collection and shrinking allow the computation to tolerate smaller caches than would otherwise be possible, with relatively little overhead. The data in Chapter 6 showed that the dense shifting traversal locks up between 9000 Kbytes and 10000 Kbyte when using only garbage collection; the data presented here shows that shrinking allows the computation to use an 8000 Kbyte cache while requiring only about 1.25 times the fastest elapsed time, or even a 4000 Kbyte cache while requiring only about 2 times the fastest elapsed time. The experiments confirm the value of garbage collection and shrinking for running with a client cache that is not large enough to hold all accessed objects simultaneously.

The MINFETCH implementation does well, as one would expect. Some of the comparisons with other policies show regions in which MINFETCH fetches more objects than the other policy: the

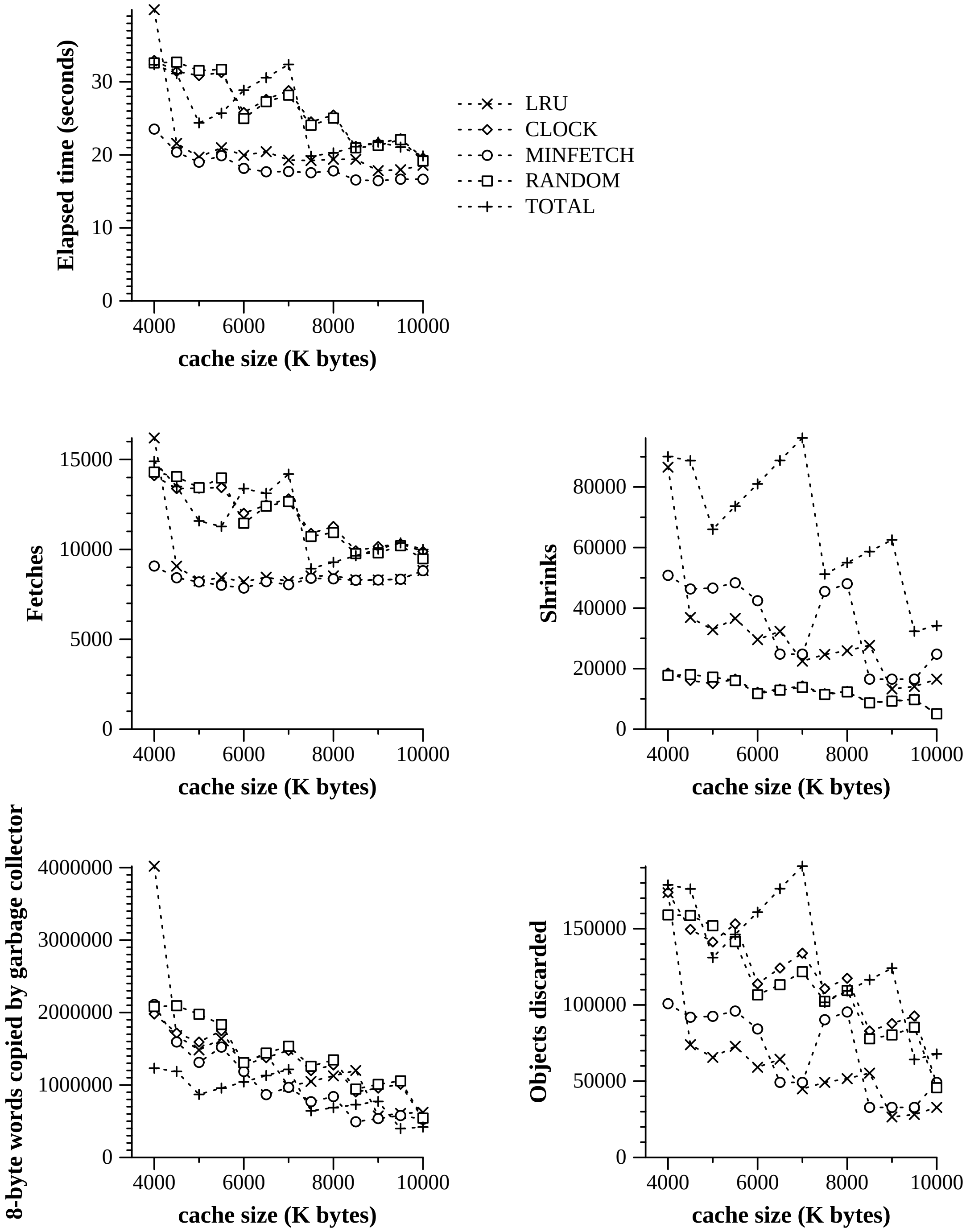

Figure 7-27: Comparing shrinking policies, detail

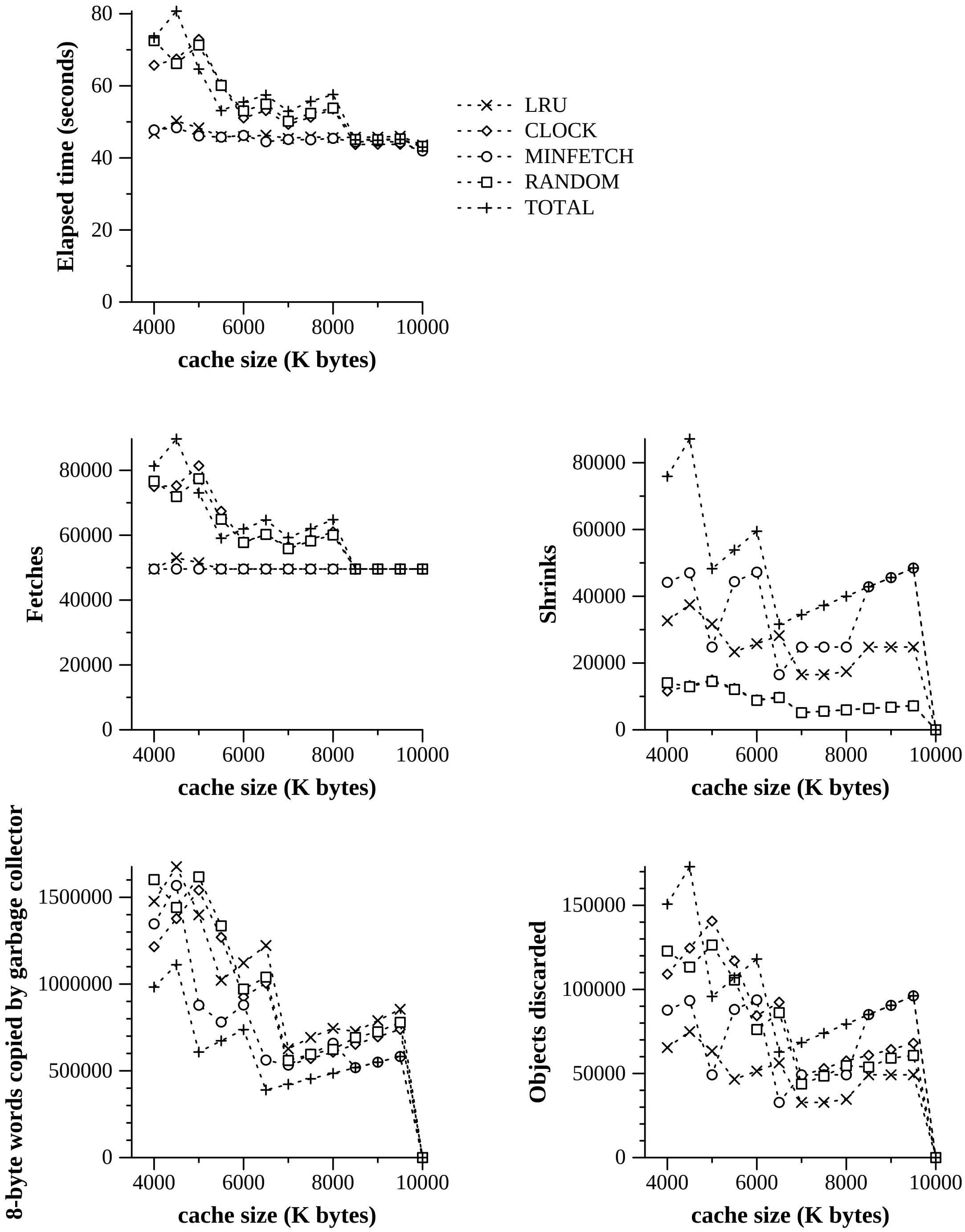

Figure 7-28: Comparing shrinking policies without prefetching

reader should keep in mind that the policies are being compared in their good configurations, not in identical configurations. Whenever the MINFETCH policy is compared to another policy on the same configuration, MINFETCH does in fact have the lowest fetch rate and the shortest elapsed time. For these traversals, the intuition seems correct that shrinking to minimize fetches is approximately the same as shrinking to minimize elapsed time.

7.7 Related Work

Chapter 6 discussed previous work on *mechanisms* like shrinking in object caching systems. There does not appear to have been any previous work on comparing shrinking *policies* in object caching systems. There are two large bodies of somewhat-related work. One is the design of processor caches, and the other is the design of virtual memory systems.

Work done on processor caches is not directly applicable to a client object cache both because processor caches typically deal with fixed-size units of transfer (cache lines) and because processor cache references occur much more frequently than object cache references. The "replacement policy" of a processor cache is typically determined by the hardware cost of building a particular degree and flavor of associativity; for recent examples, see work on column-associative caches [2] or skewed-associative caches [61]. The designers of processor caches are also concerned with how to write back changes: see Jouppi's recent work [37] for an example. This sort of work is not applicable to object caches as described in this thesis, since objects are never written back except at transaction commit.

Similarly, work done on virtual memory systems is not directly applicable to a client object cache. Again, virtual memory systems typically deal with fixed-size units of transfer (pages). The length of time a computation stays within a single page is likely to be much longer than the length of time a computation stays within a single object, especially with pages that are a hundred or more times larger than a typical object. Previous work on virtual memory policies [24] demonstrated that the fault rate of a program is much more sensitive to memory size than to the choice of paging algorithm, for all reasonable algorithms including Belady's unrealizable optimal algorithm [5]. This result is rather similar to the discovery in this chapter that the MINFETCH policy does somewhat better than any of the realizable policies, but that even its performance degrades rather quickly with decreasing memory size.

Policies like CLOCK are better than policies like RANDOM in a virtual-memory environment [24], and there are two plausible explanations for this difference. First is that objects (the granularity of eviction in Thor) are much smaller than pages (the granularity of eviction in virtual-memory systems). For the same number of different words referenced, a computation may spend longer in a single page than in a single object, with a corresponding decrease in the number of pages marked as used. The second explanation is that a virtual-memory system is typically used for both data and instruction references, whereas the client cache in Thor is used only for data references. Whatever locality there is in the code being executed does not affect the behavior of the Thor client cache, whereas it does affect a virtual memory system. When we consider the relatively good performance of LRU and the relatively poor performance of CLOCK, it seems reasonable to think in terms of the number of bits required to capture useful access patterns. One bit on a virtual memory page is apparently enough to capture some fraction of a useful access pattern for typical computations (data and instructions) in virtual memory; however, that bit is not enough to capture a useful access pattern at the granularity of data references to individual objects. Using a larger number of bits, as in the LRU implementation, improves the capture of the access pattern well enough that the system can perform quite well.

7.8 Summary

This chapter has used shifting traversals of a wide database to explore the effects of

- prefetch policy
- shrinking policy
- prefetch group size
- shrinking trigger point
- fraction of objects shrunk, and
- garbage collection trigger point

as the client cache size decreases.

In the current system, shrinking too few objects is more harmful to system performance than shrinking too many. Especially with prefetching in place, the system can recover from excessive shrinking by quickly fetching back large groups of objects. In the current Thor implementation, there is no corresponding mechanism to mitigate the expense of keeping too many objects in the heap, and repeatedly copying them. A generational collector may be a useful future addition to Thor: unshrinkable objects could migrate into an older and less-frequently-scanned generation; shrinkable objects could be shrunk at the point where they would otherwise migrate into the older generation.

The LRU policy does conspicuously better than every other policy except the unattainable MIN-FETCH policy. Among the other policies, although there are conspicuous differences in the behavior of the systems with different shrinking policies, the differences in terms of elapsed time are much less noticeable. A coarse approximation of LRU (CLOCK) is no better than simpler policies with no run-time overhead (TOTAL, RANDOM).

For the workload considered, the system's performance is relatively insensitive to changes in the trigger point or fraction of objects shrunk. The prefetch policy and prefetch group size are more important to performance than shrinking policy. When the cache is too small for all of the objects, some objects must be discarded and subsequently refetched. Because each fetch is expensive, prefetching continues to be the dominant factor in performance, as it was in Chapter 5, even though the cache was effectively infinite in the experiments described there. This is an important point of this chapter: prefetching matters a great deal, and the parameters dealing with shrinking policy are less important.

Chapter 8

Conclusion

The thesis has considered how to fetch and cache objects at clients in a distributed object database. This chapter reviews contributions and draws conclusions, and then considers future work.

8.1 Review and Conclusions

The thesis has dealt with the issues of efficiently managing objects fetched to a client cache. There were three sets of experiments presented, dealing with inter-object references, fetching objects, and evicting objects.

Chapter 4 considered the performance of two swizzling techniques (node marking and edge marking) in a single-object-fetching system. Experiments showed that edge marking is usually faster than node marking, as implemented in Thor. However, the advantage of edge marking is not large, and the real conclusion is a negative result: the choice between edge marking and node marking is not very significant. This contradicts previous work that has argued for application-specific choices of swizzling technique; that previous work was based on microbenchmarks that may have tended to exaggerate the importance of swizzling technique.

The next experiments (in Chapter 5) considered how to bring objects into the cache. Simple dynamic prefetching techniques performed better than single-object fetching. In addition, those simple dynamic techniques adapted better to a changing client workload than a technique that prefetched statically-computed page-like groups. The bf-cutoff prefetcher, which attempted to avoid sending objects already at the client, completed traversals faster than prefetchers that paid no attention to which objects had already been sent. Most of the prefetchers used only the structure of objects and the recent history of the computation in making prefetch decisions, so they would be suitable for use even when no information is available about the semantics of objects or behavior of applications. A prefetcher that made use of the semantics of objects achieved better performance than one that used only the structure of objects, but performed worse when the database changed so that it no longer matched the prefetcher's model of the objects stored.

These experiments used a narrowly-focused synthetic workload. Accordingly, it would be unreasonable to claim that dynamic prefetching is superior to static prefetching in general. However, dynamic prefetching showed important advantages with workloads where the collection of objects in use may change with different applications, with different users, or over time. While there may be databases and applications that are sufficiently static that clustering works better than dynamic prefetching, dynamic prefetching techniques seem likely to be useful for many distributed object databases and their users.

The remainder of the thesis considered how to evict objects from the cache. Chapter 6 presented the problem of cache lockup. Reliable management of an object cache requires both garbage collection and shrinking, since either technique on its own is likely to cause cache lockup. Chapter 7 compared policies for shrinking objects in the client cache. The LRU policy did nearly as well as the unattainably good MINFETCH policy, but all other policies did roughly equally poorly. Chapter 7 also put the issue of shrinking policy in context, pointing out that prefetching algorithm and prefetch group size have a larger effect on performance for the benchmark used than does the shrinking policy, shrink trigger, or shrink fraction.

For applications that do traversals of complex structures of small objects, the thesis suggests the use of edge marking and bf-cutoff prefetching, bringing over a few tens of objects at a time. An accurate least-recently-used cache management policy (like LRU) is worth implementing; however, the cheaper approximation of LRU (CLOCK) showed no particular advantage over other policies (RANDOM, TOTAL) that are easier to implement and have lower runtime overhead. Since shrinking is only used for storage management after gc has been tried, there may be quite a long time between shrinks; for the workloads used, one bit per object was not enough to capture any significant information about access patterns.

8.2 Future Work

This work has built on the best existing synthetic benchmarks for object databases and on assumptions about the important set of behaviors for applications using object databases. As more such applications are built, it is important to analyze them to determine their actual behavior, and which components of that behavior are critical to performance.

It would also be useful to extend this work, either by using new and improved synthetic benchmarks as they become available, or by covering aspects of the system that were omitted from this work. To bound the scope of the problem considered, the following concerns were omitted from this thesis: fetching objects from the server's disk, transactions that modify objects, writing objects back to the server, multiple clients fetching objects from a single server, and concurrency control problems such as conflicting transactions. There are applications for which these omitted factors limit achievable performance. It is important to understand the characteristics of such applications and the relative importance of these other factors.

There is still room for improvement in the performance of the system when fetching. The results of this thesis suggest that prefetching is the area most likely to yield performance gains. Possibilities include different prefetching algorithms or storing application-specific information with objects.

Vitter and Krishnan [72] have studied the use of data compression algorithms to achieve optimal prefetching and Curewitz, Krishnan and Vitter [20] have applied practical prefetchers derived from this theory to database traces. Their work would be a logical starting point for developing more sophisticated prefetchers to replace the simple ones that I implemented.

I have proposed a mechanism called *crystals* as a way of controlling prefetching in a system like Thor [21]. A crystal is an object that represents an explicitly- or implicitly-constructed group of objects. When the computation reaches the crystal, the associated group of objects is prefetched. The work on transparent informed prefetching [58] applies a similar approach in the context of Unix. Tait and Duchamp [70] describe a mechanism for file prefetching that attempts to work automatically, matching file access patterns to previously-seen file access patterns. It would be interesting to see if the same idea could be applied to object fetching.

The name of the method being invoked is a potentially useful piece of information that is not available to the current prefetchers. In edge marking, an object is fetched before any method is invoked upon it. (Indeed, with edge marking an object may be fetched and never used.) So the name of the method cannot be made available to the prefetcher in an edge-marking system. The name of the method could be made available in a node-marking implementation, although it is not available in the node-marking version of Thor (the current Thor implementation eliminates the original identity of the method quite early in the invocation process).

It would be useful to develop models that would allow the prediction of good prefetch group sizes. Currently both clustering and prefetching involve a certain amount of guesswork and adjustment to find a good operating point. Whatever can be done to make that more systematic will be of great value in configuring distributed object database systems.

Another suitable area for future work is building a paged version of Thor that would take advantage of fixed-size units of caching to simplify storage management. Chapter 5 showed that Thor is as good as or better than an approximation of a paged system when comparing only fetching and prefetching. A paged version of Thor could be compared to the current implementation of Thor to determine whether fetching objects allows better performance overall than fetching pages.

Bibliography

[1] Atul Adya. A distributed commit protocol for optimistic concurrency control. Master's thesis, Massachusetts Institute of Technology, February 1994.

[2] Anant Agarwal and Steven D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190, 1993.

[3] T. Lougenia Anderson, Arne J. Berre, Moira Mallison, Harry H. Porter, III, and Bruce Schneider. The HyperModel benchmark. In *Conference on Extended Database Technology (EDBT 90)*, pages 317–331. Springer-Verlag, 1990.

[4] François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database: The Story of O_2*. Morgan Kaufmann, 1992.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[6] Véronique Benzaken and Claude Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice*, pages 403–412. Morgan Kaufmann, 1991.

[7] Margaret H. Butler. Storage reclamation in object oriented database systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 410–425, 1987.

[8] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[9] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS extensible DBMS project: An overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann, 1990.

[10] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C.K. Tan, Odysseas G. Tsatalos, Seth White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 383–394, 1994.

[11] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–426, 1994.

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

[13] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis. Fine-grained sharing in a page server OODBMS. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 359–370, 1994.

[14] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[15] 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 ACM SIGMOD Conference on Management of Data*, pages 348–357, 1989.

[16] Jia-bing R. Cheng and A. R. Hurson. Effective clustering of complex objects in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 22–31, 1991.

[17] Jia-bing R. Cheng and A. R. Hurson. On the performance issues of object-based buffering. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, pages 30–37, 1991.

[18] H.-T. Chou, D. DeWitt, R. Katz, and A. Klug. Design and implementation of the Wisconsin storage system. *Software: Practice and Experience*, 15(10):943–962, October 1985.

[19] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 43–52, 1992.

[20] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. Brown University Technical Note.

[21] Mark Day. Object groups may be better than pages. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 119–122, 1993.

[22] Mark Day, Sanjay Ghemawat, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Theta reference manual. Memo 88, MIT LCS Programming Methodology Group, December 1994.

[23] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[24] Peter J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, September 1970.

[25] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, January 1980.

[26] O. Deux. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[27] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Velez. A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the 16th Conference on Very Large Data Bases*, pages 107–121, Brisbane, Australia, 1990.

[28] Pamela Drew and Roger King. The performance and utility of the Cactis implementation algorithms. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 135–147, 1990.

[29] Sanjay Ghemawat. Disk management for object-oriented databases. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 222–225, 1993.

[30] Sanjay Ghemawat, M. Frans Kaashoek, and Barbara Liskov. Disk management policies for object-oriented databases. Extended abstract submitted to OSDI '94.

[31] Olivier Gruber and Laurent Amsaleg. Object grouping in Eos. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*, pages 117–131. Morgan Kaufmann, San Mateo, California, 1994.

[32] John L. Hennessey and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[33] Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 288–303, 1993.

[34] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 106–119, 1993.

[35] Tony Hosking and Eliot Moss, March 1994. Personal communication.

[36] Hiroshi Ishikawa, Fumio Suzuki, Fumihiko Kozakura, Akifumi Makinouchi, Mika Miyagishima, Yoshio Izumida, Masaaki Aoshima, and Yasuo Yamane. The model, language, and implementation of an object-oriented multimedia knowledge base management system. *ACM Transactions on Database Systems*, 18(1):1–50, March 1993.

[37] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, 1993.

[38] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[39] Ted Kaehler and Glenn Krasner. LOOM – Large object-oriented memory for Smalltalk-80 systems. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 298–307. Morgan Kaufmann, 1990.

[40] Alfons Kemper and Donald Kossmann. Adaptable pointer swizzling strategies in object bases. In *Proceedings of the 9th International Conference on Data Engineering*, pages 155–162, 1993.

[41] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Darrell Woelk, and Jay Banerjee. Integrating an object-oriented programming system with a database system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 142–152, 1988.

[42] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.

[43] Butler Lampson and Howard Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, 1979.

[44] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[45] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22, 1976.

[46] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*, pages 79–91. Morgan Kaufmann, San Mateo, California, 1994.

[47] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 226–238, October 1991.

[48] Umesh Maheshwari. Distributed garbage collection in a client-server, transactional, persistent object system. Technical Report MIT/LCS/TR-574, MIT Laboratory for Computer Science, 1993.

[49] Salvatore T. March. Techniques for structuring database records. *Computing Surveys*, 15(1):45–79, March 1983.

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

[51] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[52] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. Technical Report 90-38, COINS, University of Massachusetts - Amherst, 1990.

[53] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[54] Andrew Myers. Personal communication, June 1994.

[55] Andrew C. Myers. Fast object operations in a persistent programming system. Technical Report MIT/LCS/TR-599, MIT Laboratory for Computer Science, January 1994.

[56] James O'Toole. Garbage collecting the object cache. An informal working document., October 1992.

[57] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, 1991.

[58] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. Technical Report CMU-CS-93-113, Computer Science Department, Carnegie Mellon University, February 1993.

[59] Peter Scheuermann, Young Chul Park, and Edward Omiecinski. Heuristic reorganization of clustered files. In W. Litwin and H.-J. Schek, editors, *Proceedings of the 3rd International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 16–30. Springer-Verlag, 1989.

[60] Mario Schkolnick. A clustering algorithm for hierarchical structures. *ACM Transactions on Database Systems*, 2(1):27–44, March 1977.

[61] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, 1993.

[62] Karen Shannon and Richard Snodgrass. Semantic clustering. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice.*, pages 389–402. Morgan Kaufmann, 1991.

[63] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, 1992.

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

[65] Alan Snyder. The essence of objects: Concepts and terms. *IEEE Software*, pages 31–42, January 1993.

[66] Marc San Soucie, Allen Otis, and Bob Bretl, October 1993. Personal communication in response to an early draft of the section on GemStone.

[67] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[68] James William Stamos. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Technical Report SCG-82-2, Xerox Palo Alto Research Center, May 1982.

[69] Andrew Straw, Fred Mellender, and Steve Riegel. Object management in a persistent Smalltalk system. *Software – Practice and Experience*, 19(8):719–737, August 1989.

[70] Carl D. Tait and Dan Duchamp. Detection and exploitation of file working sets. In *Proceedings of the 11th Conference on Distributed Computing Systems*, pages 2–9, 1991.

[71] Manolis M. Tsangaris and Jeffrey F. Naughton. On the performance of object clustering techniques. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 144–153, 1992.

[72] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 121–130, 1991.

[73] Seth J. White and David J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the 18th VLDB Conference*, pages 419–431, 1992.

[74] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management*, pages 1–42. Springer-Verlag, 1992. Lecture Notes in Computer Science 637.

[75] C.T. Yu, Cheing-Mei Suen, K. Lam, and M.K. Siu. Adaptive record clustering. *ACM Transactions on Database Systems*, 10(2):180–204, June 1985.

Appendix A

The minfetch algorithm

The MINFETCH shrinking policy uses knowledge of the future computation to minimize the number of fetches. Since fetches are the most expensive part of a traversal, a policy that minimizes fetches is likely to minimize elapsed time.

This appendix presents the full version of the MINFETCH algorithm for a cache that may contain unshrinkable objects. First, though, Figure A-1 shows the simpler form that is possible when dealing entirely with persistent, unmodified objects; this simpler form is the one used in the experiments reported here.

The MINFETCH algorithm ensures the following invariant: For all objects x in the cache at the end of MINFETCH, there does not exist any object y outside the cache such that y was in the cache before MINFETCH ran and the next use of y occurs before the next use of x in the continuation of the computation.

The algorithm minimizes the number of fetches required. This problem is not solved by Belady's OPT algorithm [5] for paging systems, since OPT only deals with a single page at a time; the MINFETCH algorithm discards multiple objects of varying sizes. Since a paged system can only discard whole

1. Ensure that a garbage collection has run, so that all objects in the cache are reachable.
2. Use a *future cache* identical in size to the real cache. However, the future cache is empty.
3. Use perfect knowledge of the future computation to simulate continuing the computation: for each object used in the future, add it to the future cache if it is not already in the future cache.
4. When the future cache is full or the computation ends, compare the existing cache and future cache. Discard any persistent objects that are in the old cache but not in the new cache.

Figure A-1: The simple MINFETCH algorithm

1. Ensure that a garbage collection has run, so that all objects in the cache are reachable.
2. Use a *future cache* identical in size to the real cache. However, the future cache is empty.
3. Add all non-persistent objects to the future cache. None of them can be shrunk. They will be eliminated later if possible.
4. Use perfect knowledge of the future computation to simulate continuing the computation: for each object used in the future, add it to the future cache if it is not already in the future cache.
5. When the future cache is full, run a garbage collection on the future cache. If some objects are unreachable, so that the future cache is no longer full, then repeat the algorithm from that point in the future. Treat the future cache as the existing cache and allocate a new future cache.
6. When the future cache is full and remains full after GC, or the computation ends, compare the existing cache and future cache. Discard any persistent objects that are in the old cache but not in the new cache.

Figure A-2: The full MINFETCH algorithm

pages, there are configurations in which MINFETCH does better than Belady's OPT at minimizing fetches. However, Belady's OPT can never do better than MINFETCH given the same knowledge, because MINFETCH can always discard or fetch the same objects that OPT does.

For completeness, Figure A-2 shows the full MINFETCH algorithm to be used when some objects may be unshrinkable.

Appendix B

Storage Management Data

This appendix contains the full set of graphs for different configurations of the storage managers.

B.1 MINFETCH shrinking policy

B.1.1 Varying GC trigger

Figure B-1 shows the effect of decreasing the gc trigger to 2000 when using the MINFETCH policy. Decreasing the gc trigger decreases the frequency of garbage collection, which should improve performance. As was true for TOTAL, a gc trigger of 2000 performs very slightly better than a gc trigger of 20000. However, again as was true for TOTAL, a further decrease to 1000 causes the system to run out of storage for some cache sizes.

Figure B-2 shows the effect of increasing the gc trigger to 50000 when using the MINFETCH policy. Increasing the gc trigger increases the frequency of garbage collection, which should decrease performance. As with TOTAL, the graphs show a very small degradation of performance.

B.1.2 Varying Prefetch Algorithm

Figure B-3 shows the effect of varying the prefetch algorithm when using the MINFETCH policy. As with the TOTAL policy, bf-cutoff has the best performance of the three practical algorithms, even though bf-continue sends fewer fetch requests to the server. As previously observed for TOTAL, most of the curves for bf-continue are very similar to the corresponding curves for bf-cutoff, but the bf-continue curves are shifted to the right. This means that with bf-continue, the effective cache size is smaller than it is with bf-cutoff prefetching.

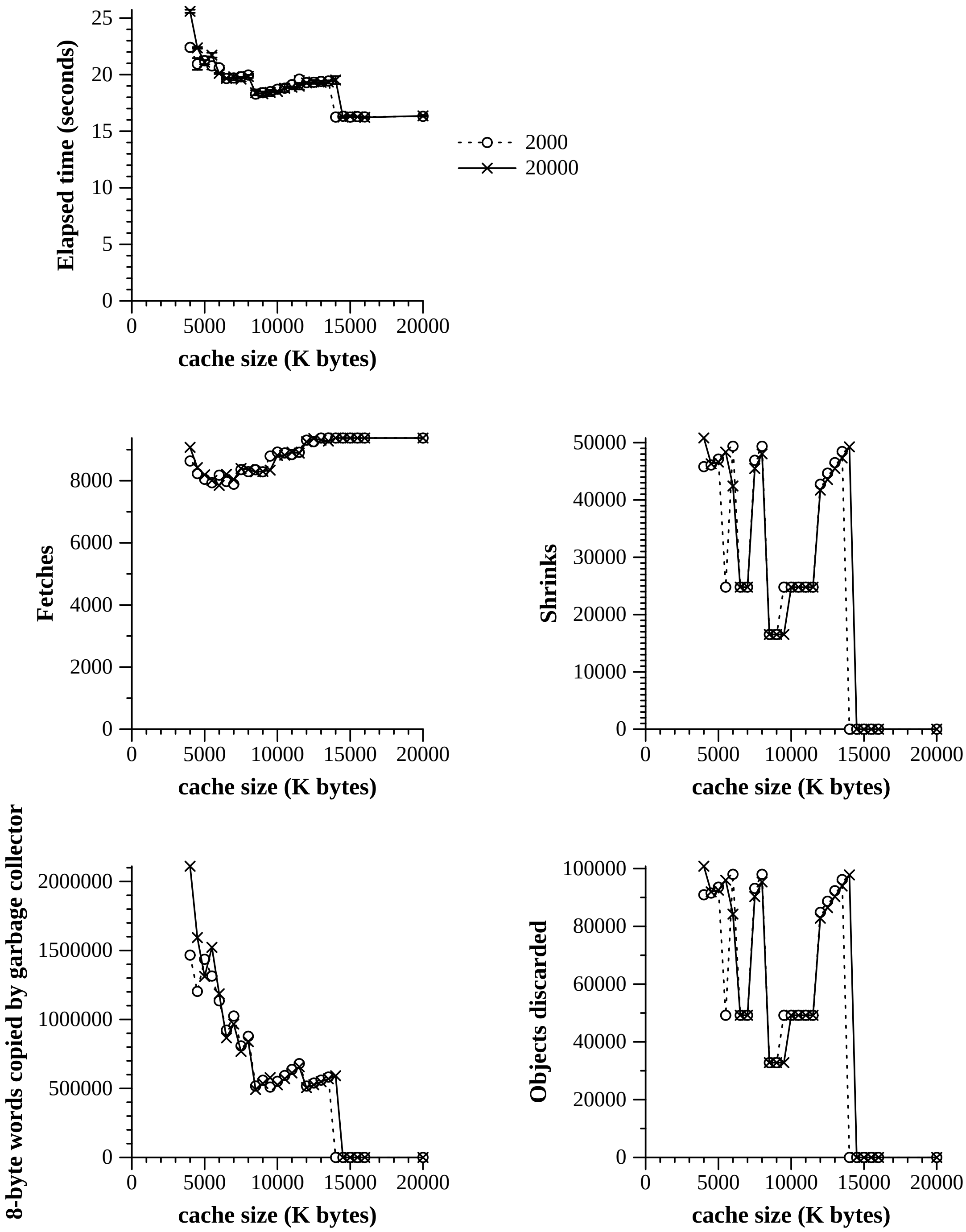

Figure B-1: MINFETCH, decreasing gc trigger

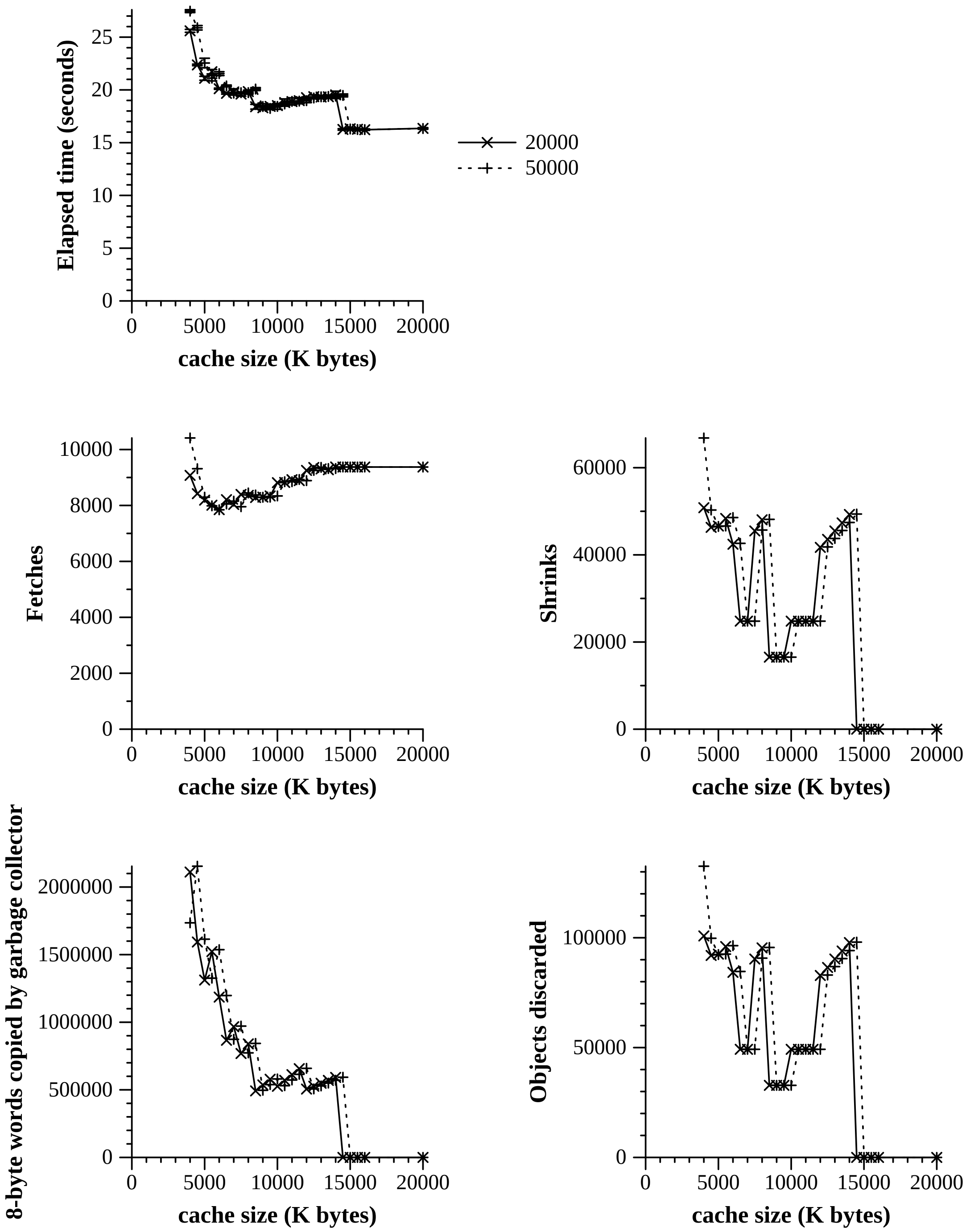

Figure B-2: MINFETCH, increasing gc trigger

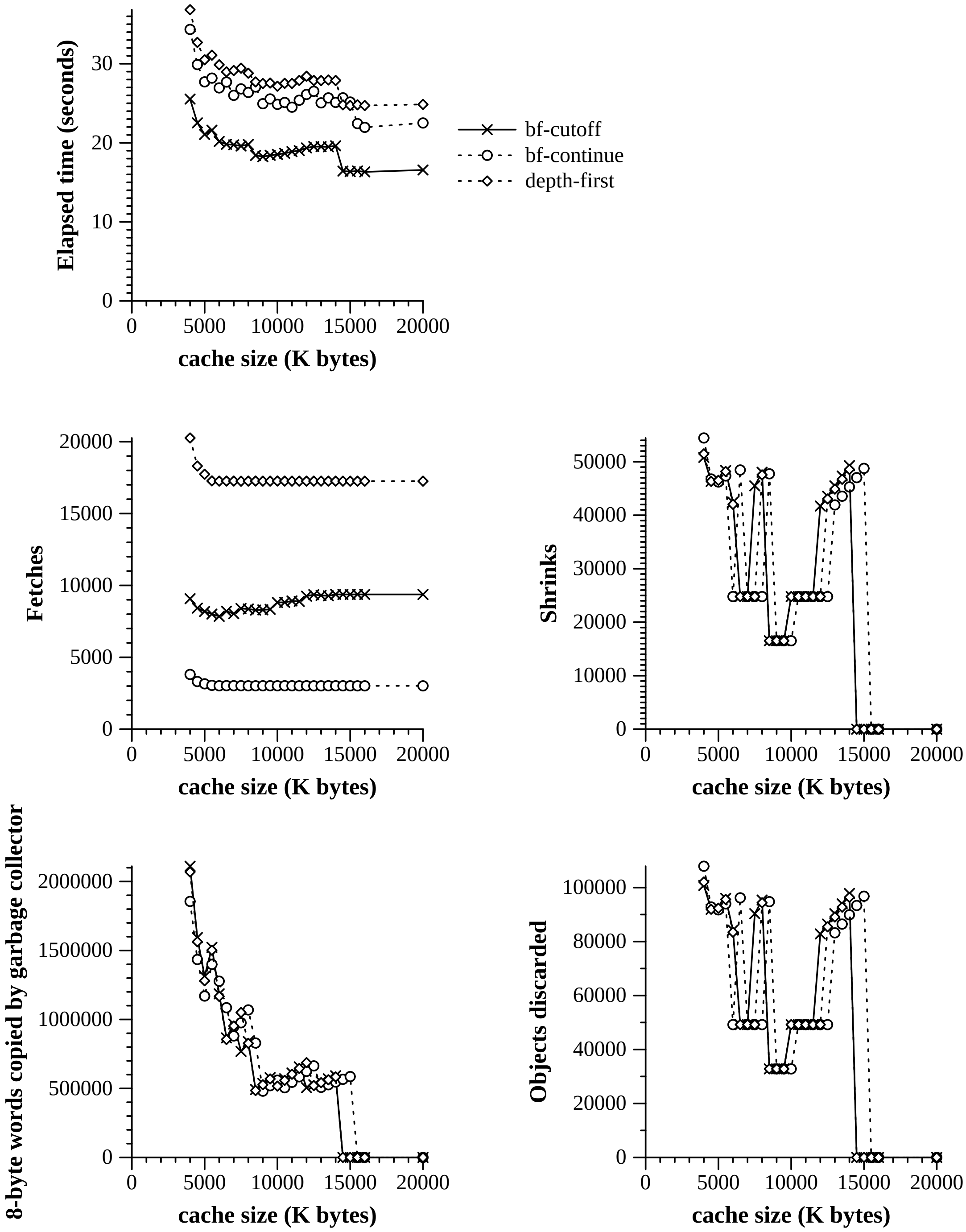

Figure B-3: MINFETCH, varying prefetch algorithm

B.1.3 Varying Prefetch Group Size

Figure B-4 shows the effect of reducing the maximum prefetch group size when using the MINFETCH policy. Figure B-5 shows the effect of increasing the maximum prefetch group size when using the MINFETCH policy. Most of the curves look quite similar to data examined earlier. As with TOTAL, reducing the maximum prefetch group size has a large effect on performance. Increasing the maximum prefetch group size has comparatively little effect on performance, although there is a notable volatility to the system's behavior with a maximum prefetch group size of 75.

B.1.4 Varying Shrink Trigger

Figure B-6 shows the effect of varying the shrink trigger when using the MINFETCH policy. The data is quite similar to the corresponding data for the TOTAL policy: it is clear that a shrink trigger of 0.05 performs poorly due to not shrinking enough objects. The other two curves represent broad ranges of shrink trigger values: one from roughly 0.25 to 0.35, the other from roughly 0.45 up. The curves for 0.40 (not shown) are identical to those for 0.45+ except for slightly worse performance at very small cache sizes.

The lower region (0.25 – 0.35) clearly represents better performance than the upper region (0.45+). We thus know that reducing the shrink trigger to the range of 0.25 – 0.35 is better than its default value of 0.5, but that reducing it to 0.05 is worse.

Figure B-7 shows the effect of shrink trigger values smaller than 0.25 but larger than 0.05. Again as with TOTAL, there is no advantage to reducing the shrink trigger below 0.25.

B.2 CLOCK shrinking policy

B.2.1 Varying GC trigger

Figure B-8 shows the effect of varying the gc trigger for a system using CLOCK (with a shrink fraction of 0.15 instead of 0.5). The conclusion is familiar from the discussion of TOTAL and MINFETCH in earlier sections: Varying the gc trigger has little effect until it gets too small. A gc trigger of 1000 is too small to finish the traversal.

B.2.2 Varying maximum prefetch group size

Figure B-9 shows the effect of reducing the maximum prefetch group size for a system using CLOCK (with a shrink fraction of 0.15 instead of 0.5). Figure B-10 shows the effect of increasing the maximum prefetch group size. The qualitative effect is very similar to what has been shown before for variations in prefetch group size:

- reducing the group size has a large effect, increasing it has a small effect;

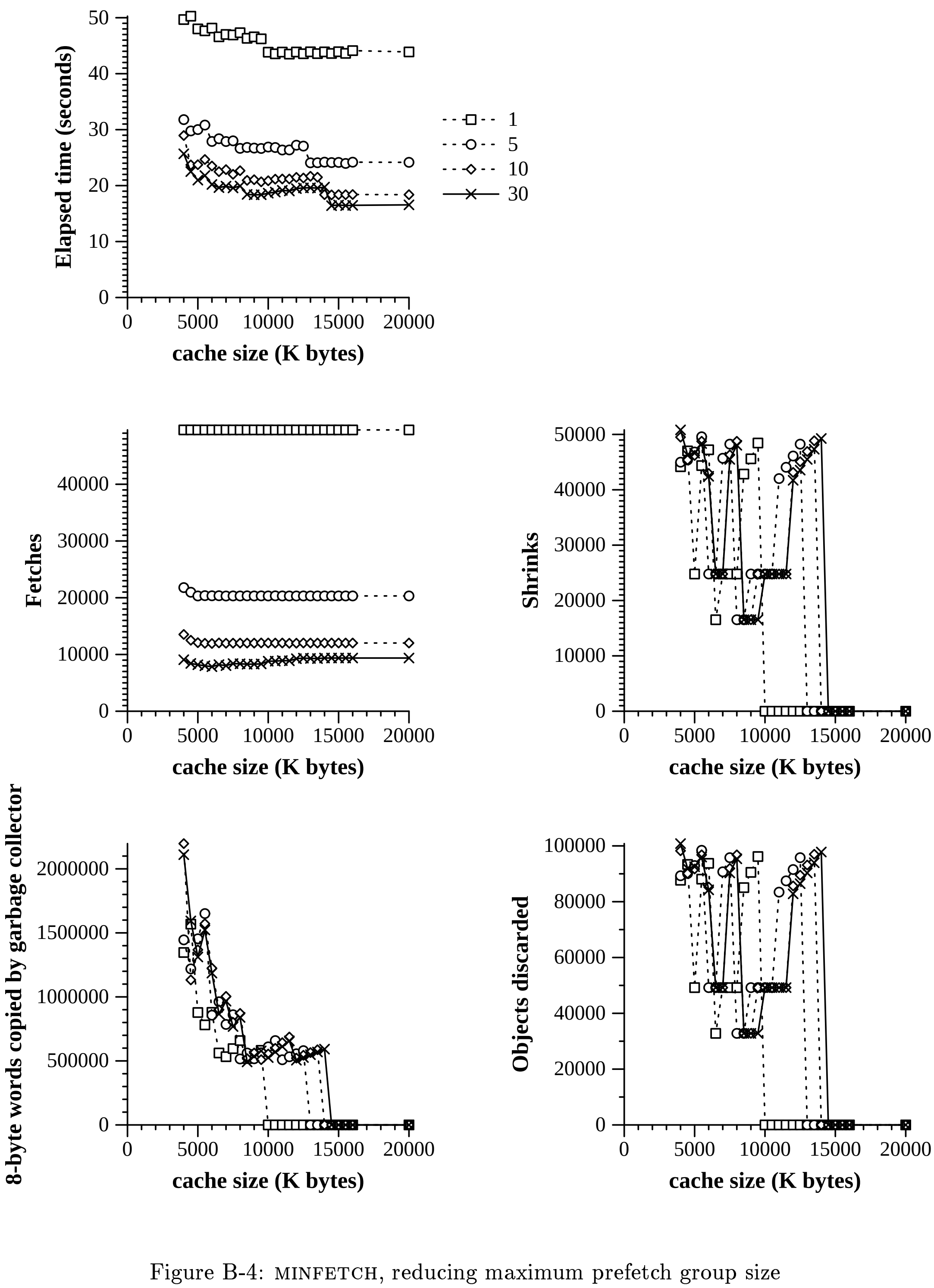

Figure B-4: MINFETCH, reducing maximum prefetch group size

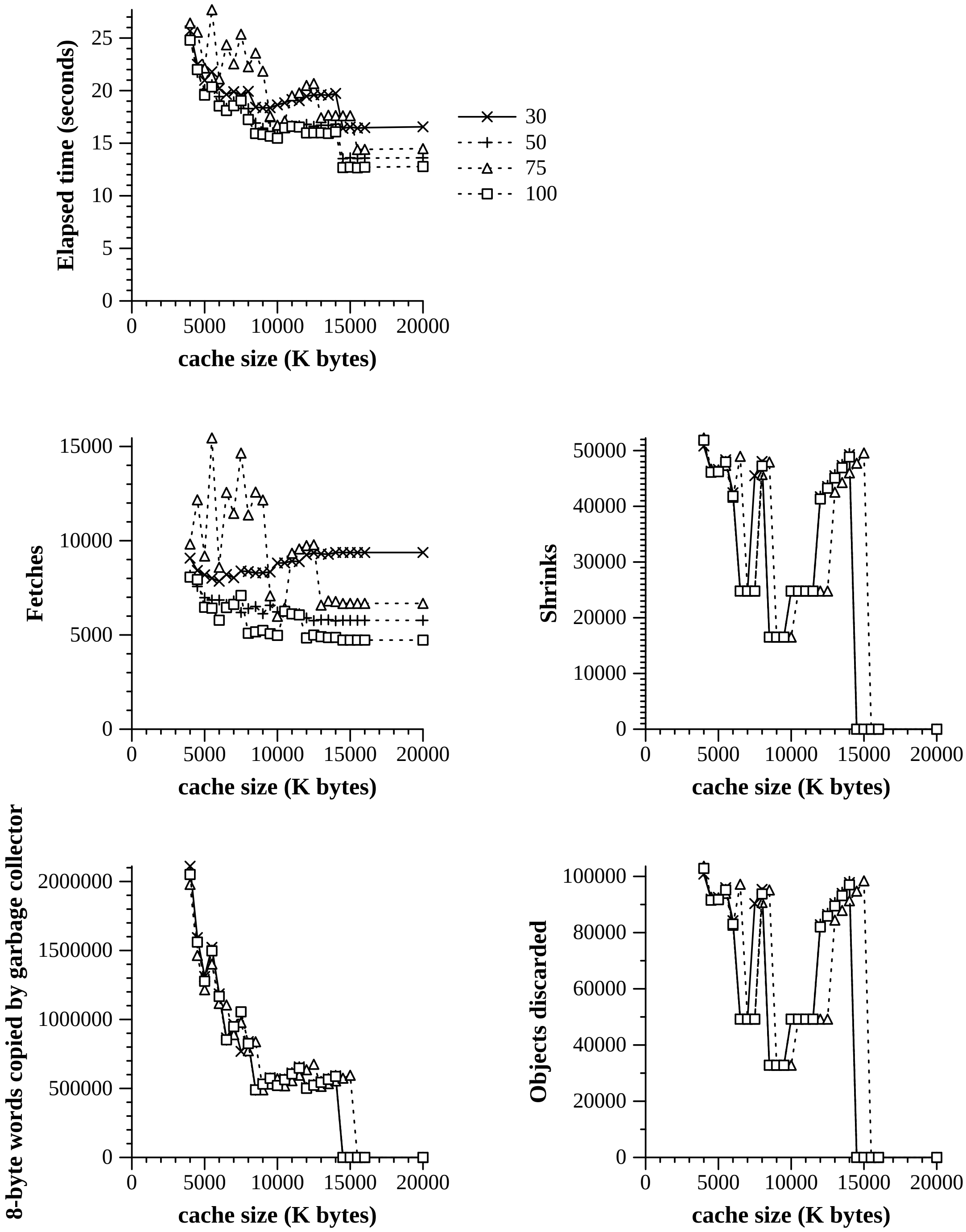

Figure B-5: MINFETCH, increasing maximum prefetch group size

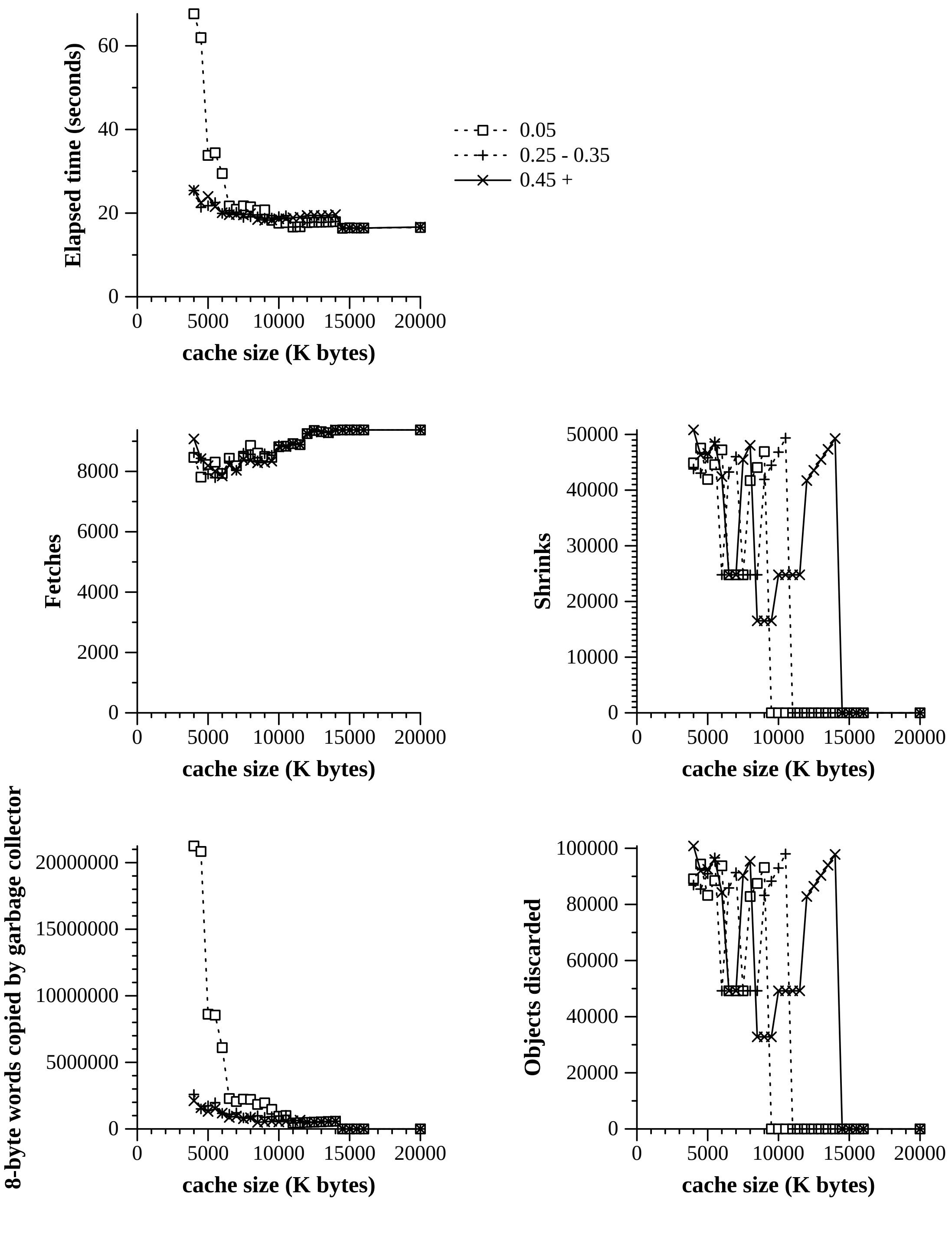

Figure B-6: MINFETCH, varying shrink trigger

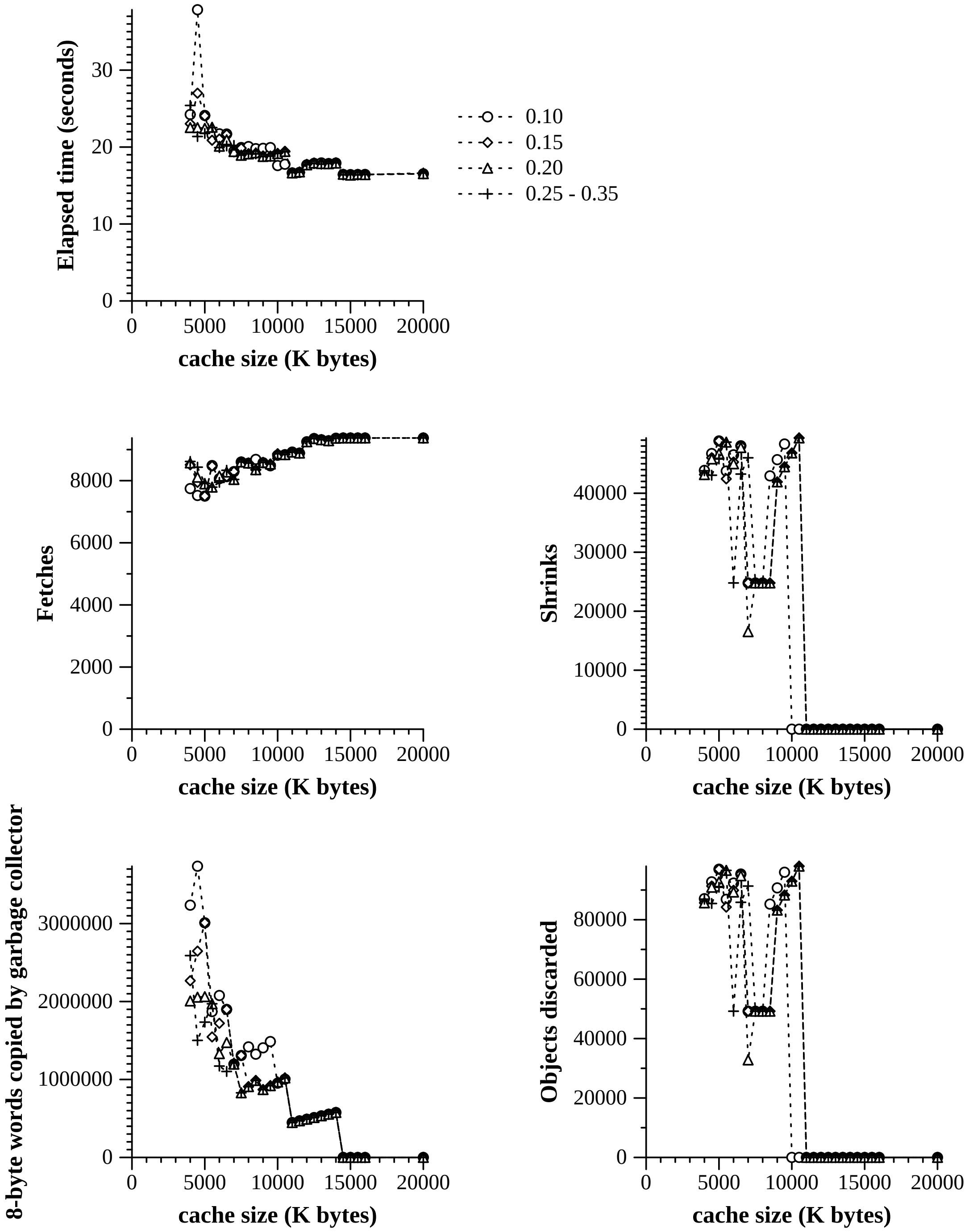

Figure B-7: MINFETCH, smaller shrink triggers

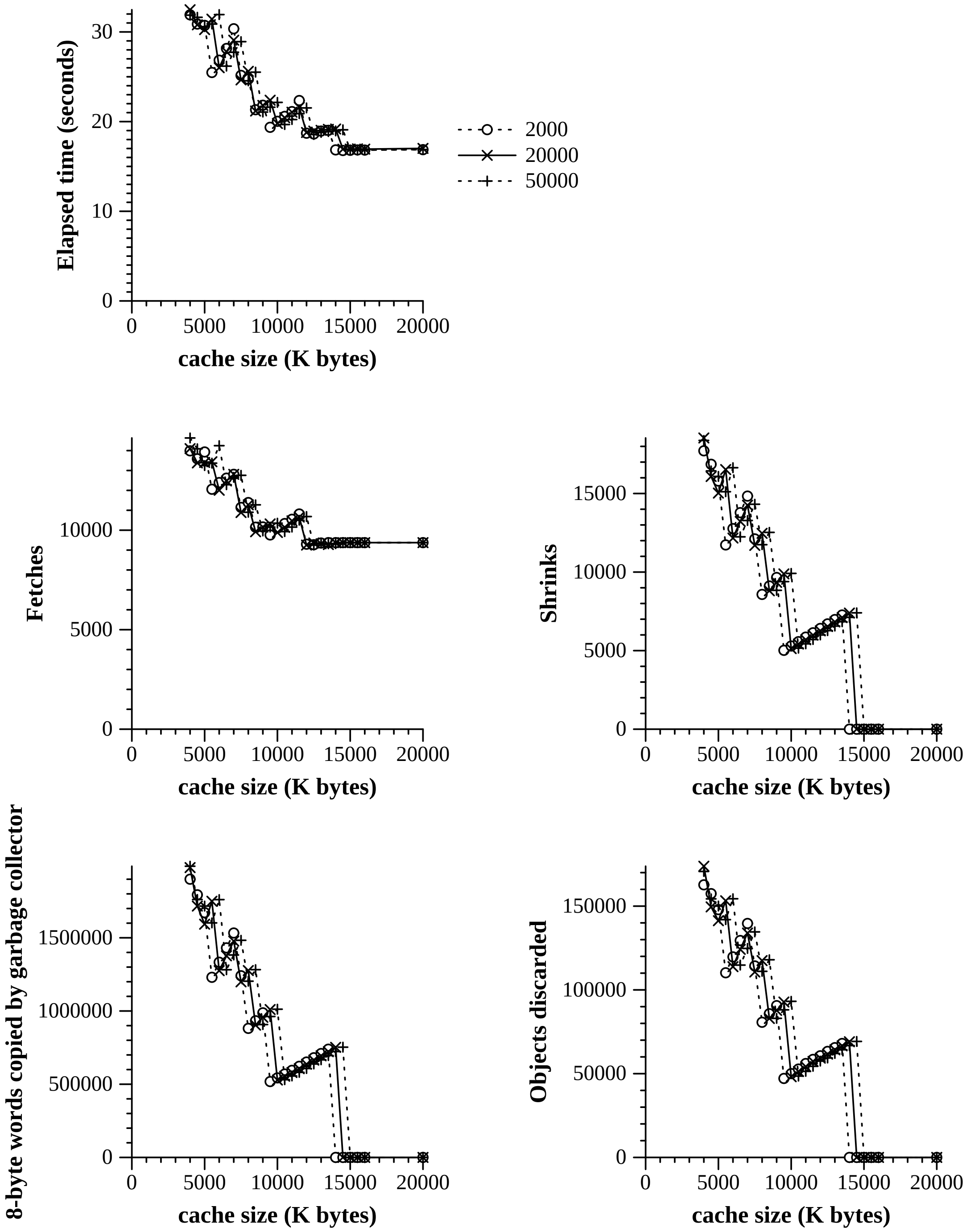

Figure B-8: CLOCK, small shrink fraction, varying gc trigger

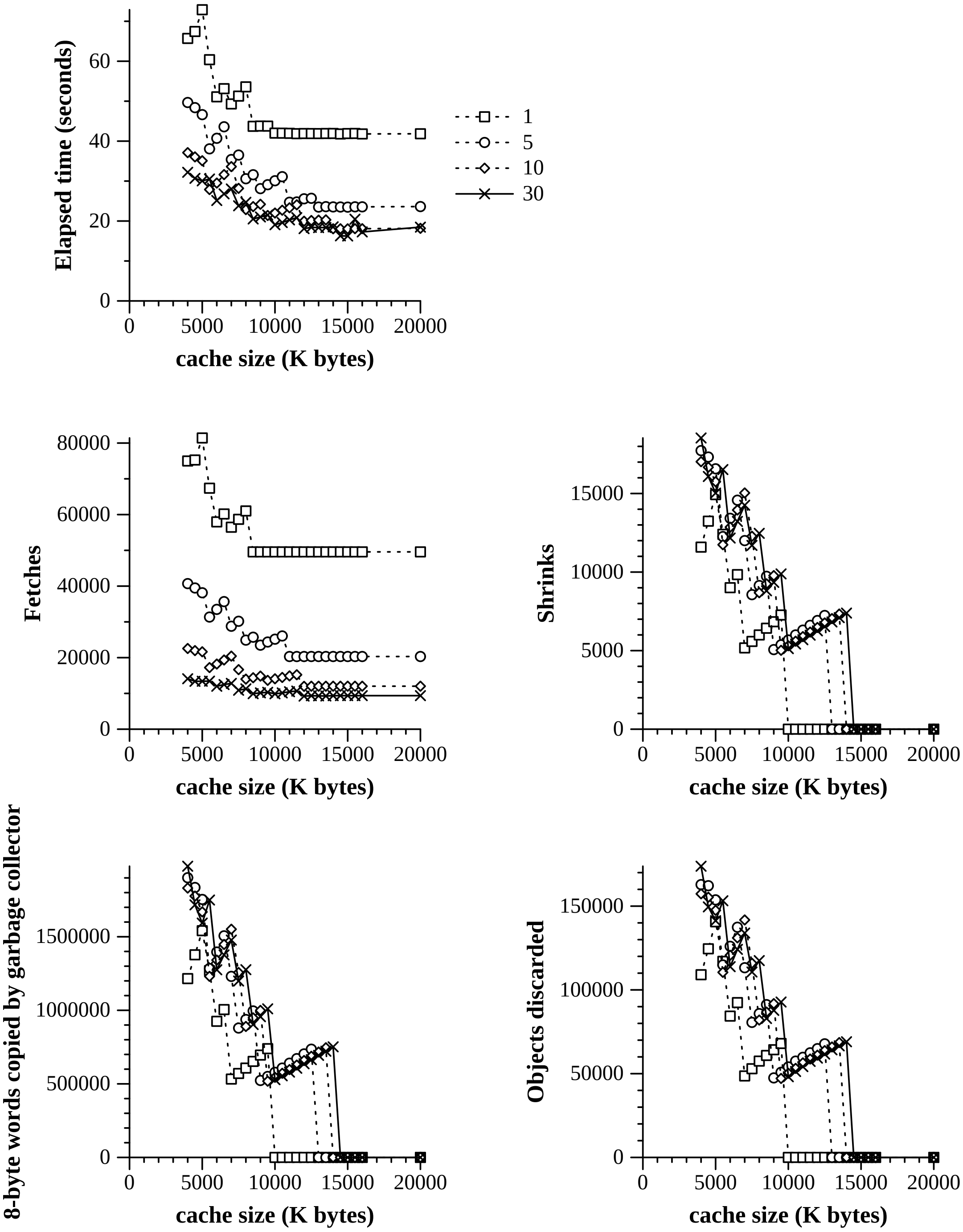

Figure B-9: CLOCK, small shrink fraction, reducing maximum prefetch group

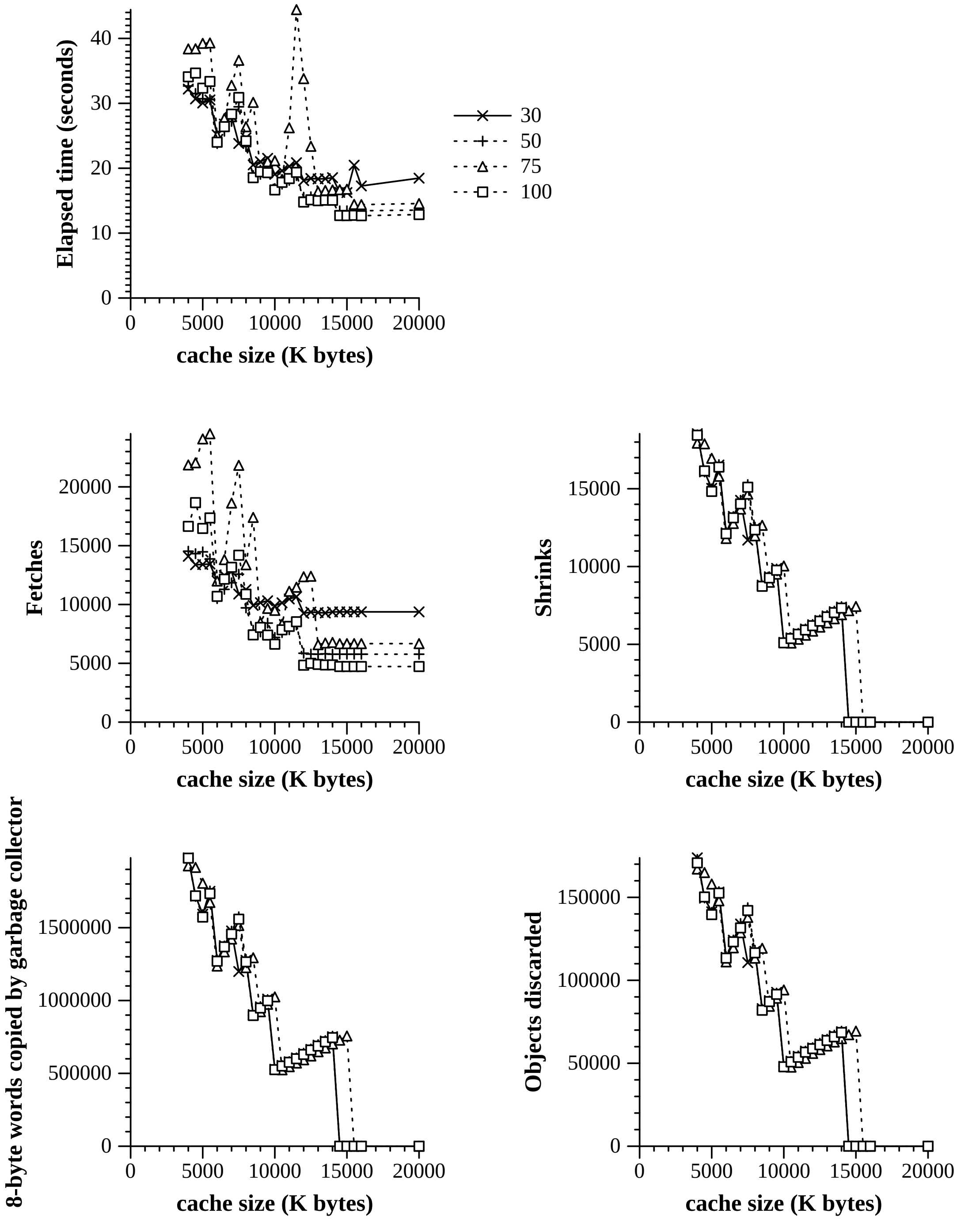

Figure B-10: CLOCK, small shrink fraction, increasing maximum prefetch group

- there is a noticeable increase in the volatility of the system for a prefetch batch size of 75.

B.2.3 Varying prefetching algorithm

Figure B-11 shows the effect of varying the prefetching algorithm for a system using CLOCK (with a shrink fraction of 0.15 instead of 0.5). As has been observed for other configurationso and other shrinking policies, bf-cutoff prefetching has the lowest elapsed time even though bf-continue causes fewer fetches.

B.3 RANDOM shrinking policy

B.3.1 Varying GC trigger

Figure B-12 shows the performance of RANDOM with a small shrink fraction as the gc trigger is varied. There is nothing surprising here: again, there is little performance difference until the gc trigger gets too small, at which point the traversal cannot complete.

B.3.2 Varying maximum prefetch group size

Figure B-13 shows the effect of reducing the maximum prefetch group size for a system using RANDOM (with a shrink fraction of 0.15 instead of 0.5). Figure B-14 shows the effect of increasing the maximum prefetch group size. The qualitative effect is very similar to what has been shown before for variations in prefetch group size:

- reducing the group size has a large effect, increasing it has a small effect;
- there is a noticeable increase in the volatility of the system for a prefetch batch size of 75.

B.3.3 Varying prefetching algorithm

Figure B-15 shows the effect of varying the prefetching algorithm for a system using RANDOM (with a shrink fraction of 0.15 instead of 0.5). As has been observed for other configurationso and other shrinking policies, bf-cutoff prefetching has the lowest elapsed time even though bf-continue causes fewer fetches.

B.4 LRU shrinking policy

Figure B-16 compares LRU to CLOCK for the default configuration. It is already apparent from this comparison that LRU generally offers more consistent performance than CLOCK although it seems to thrash more at small cache sizes (as shown by its performance with a cache size of 4000 Kbytes). It

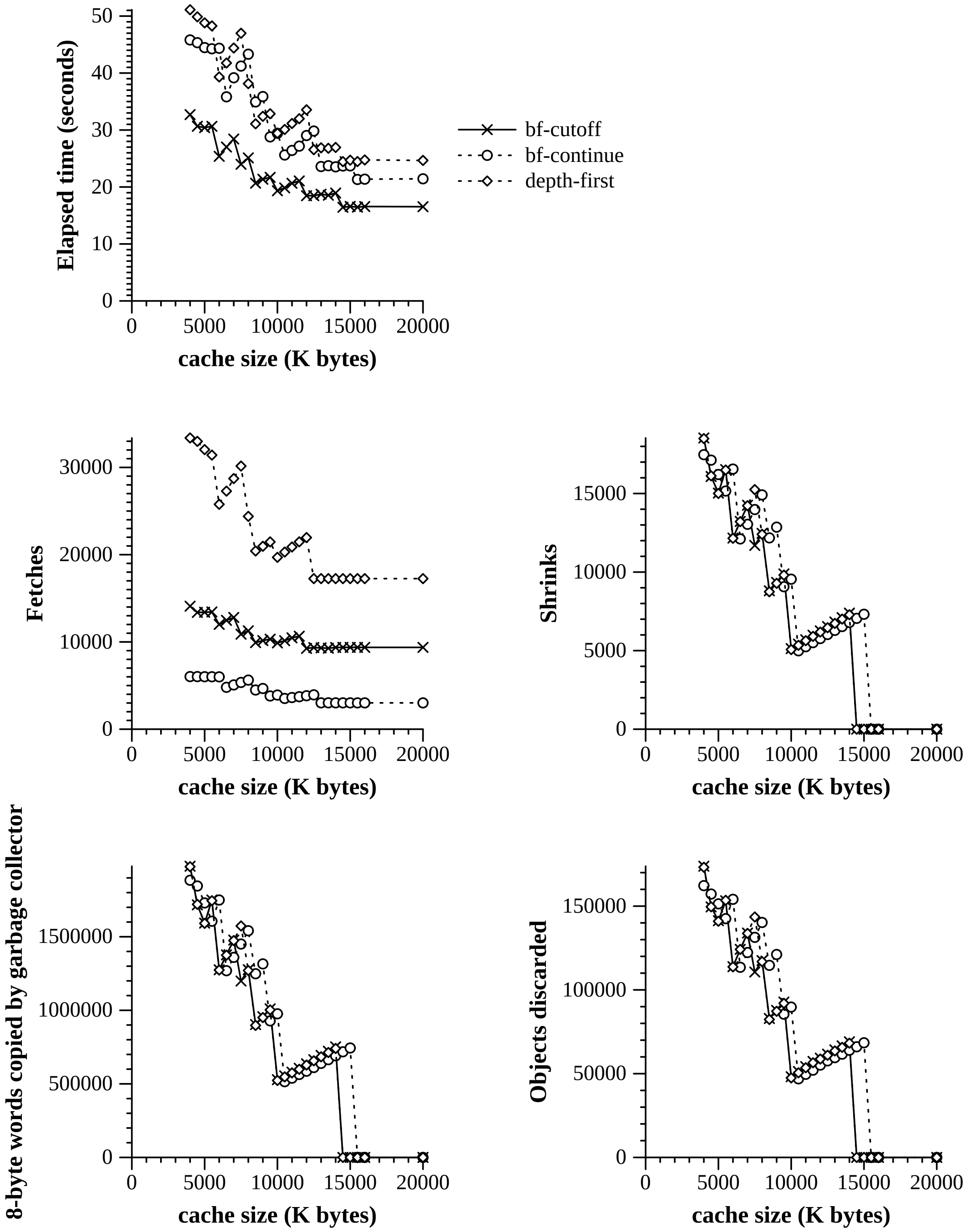

Figure B-11: CLOCK, small shrink fraction, varying prefetch algorithm

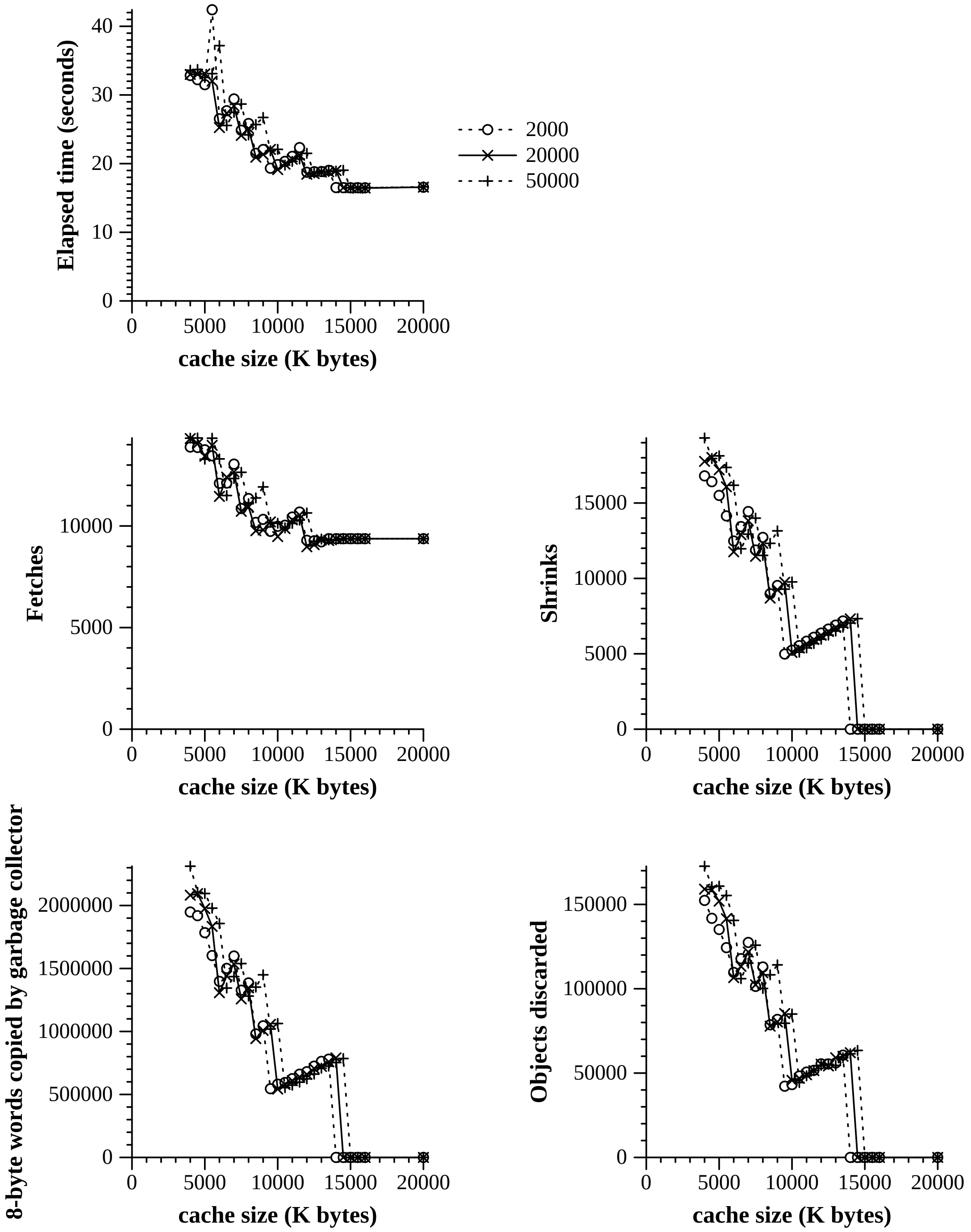

Figure B-12: RANDOM, small shrink fraction, varying GC trigger

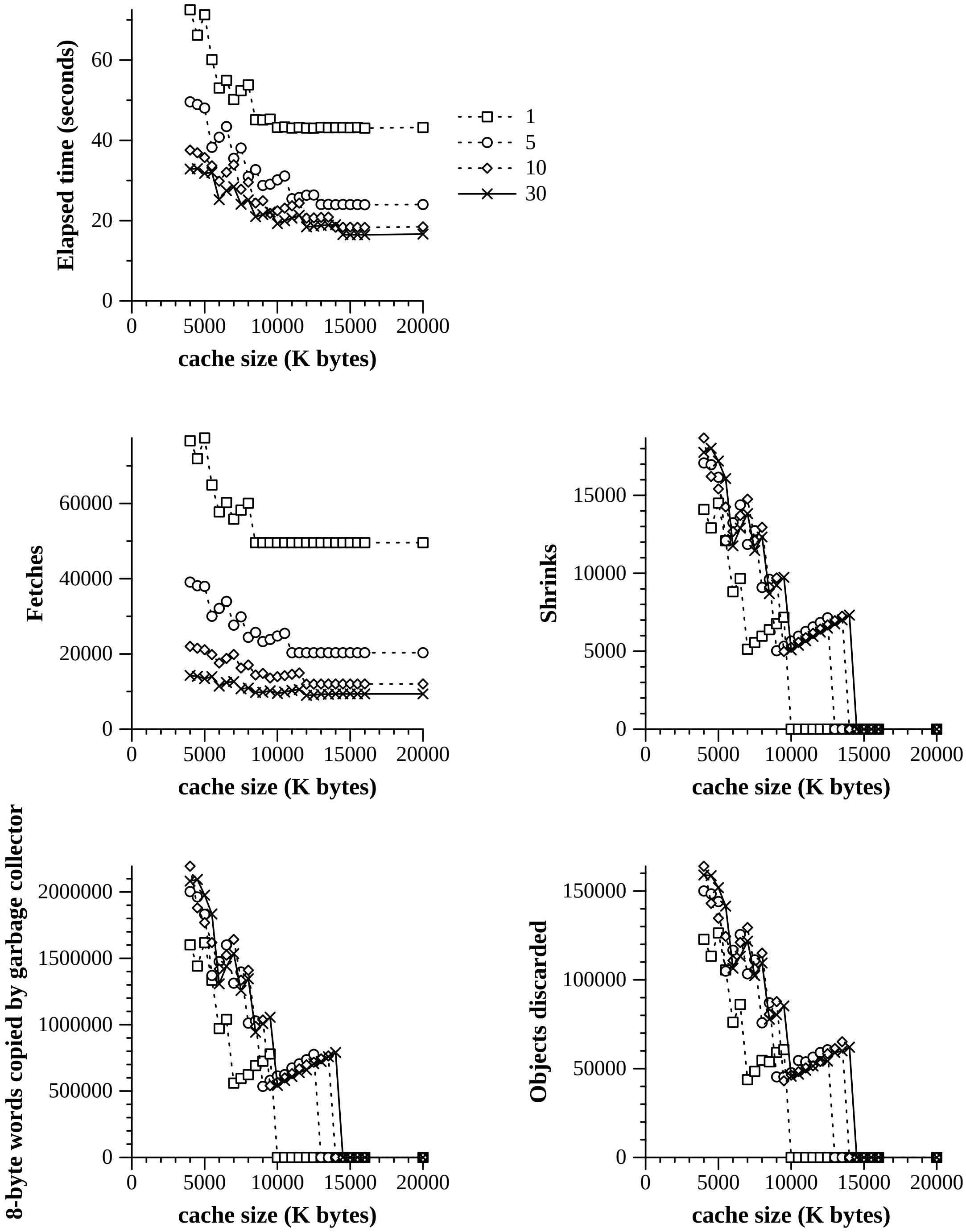

Figure B-13: RANDOM, small shrink fraction, reducing maximum prefetch group

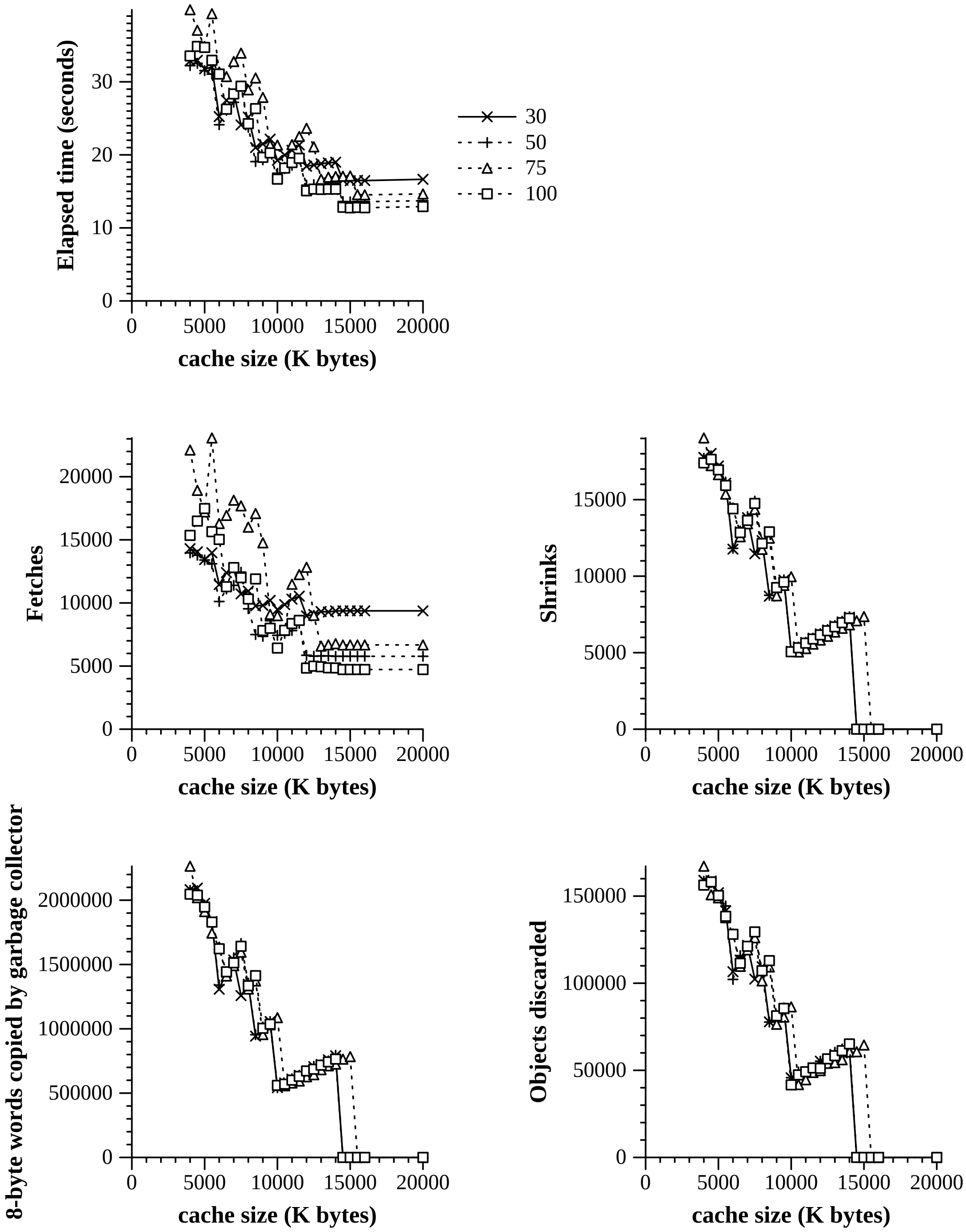

Figure B-14: RANDOM, small shrink fraction, increasing maximum prefetch group

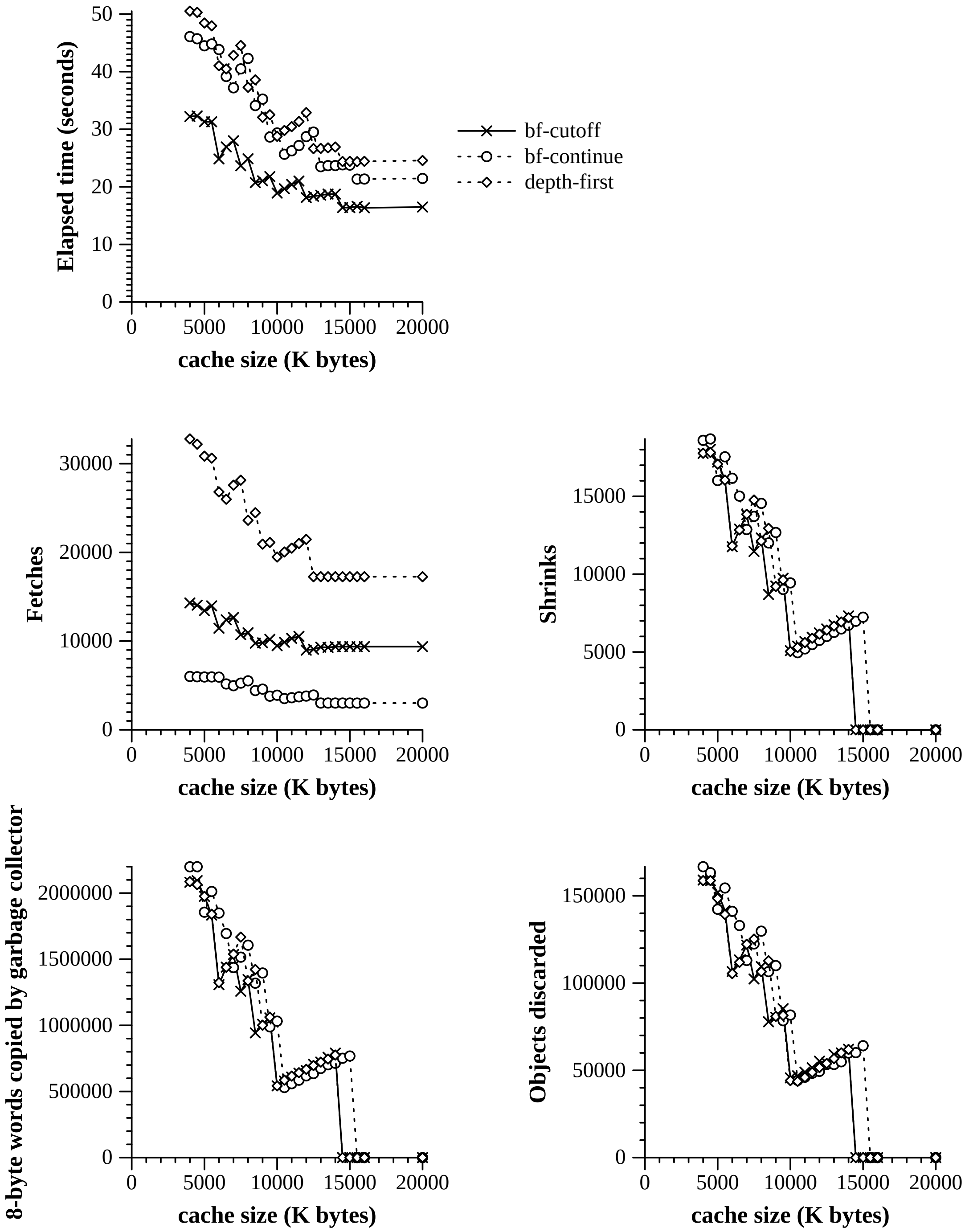

Figure B-15: RANDOM, small shrink fraction, varying prefetch algorithm

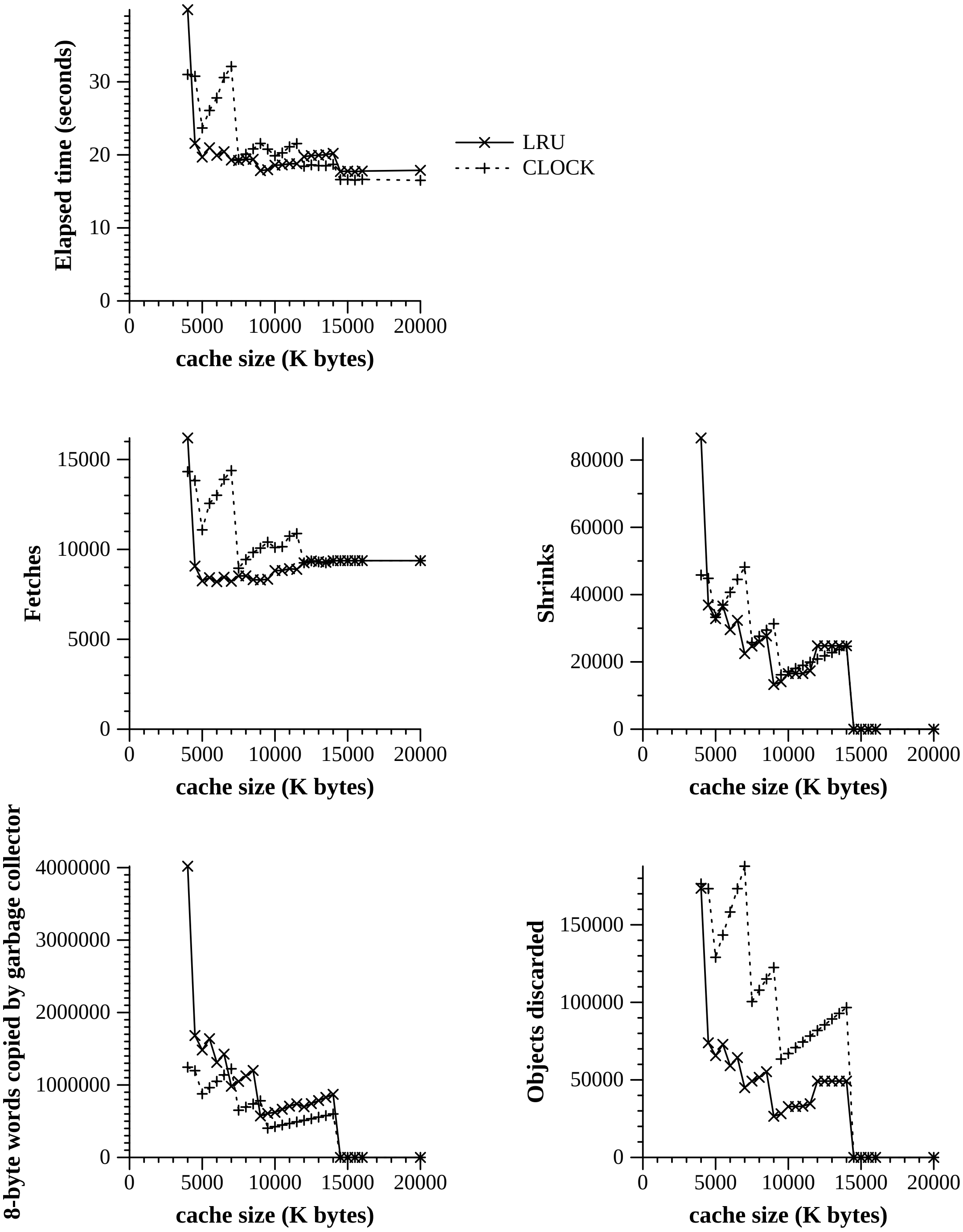

Figure B-16: LRU compared to CLOCK

is also noteworthy that although LRU and CLOCK are generally shrinking roughly the same number of objects, LRU actually discards many fewer objects: this seems to indicate that the LRU policy is more successful than CLOCK at choosing "leaf" objects that have become irrelevant, and that CLOCK is choosing some objects that are internal to the tree being traversed, causing child objects to be discarded.

B.4.1 Varying the shrink fraction

Figure B-17 shows the effect of reducing the shrink fraction. These graphs show that reducing the shrink fraction only affects the elapsed time noticeably for caches below 10000 Kbytes.

Figure B-18 shows the details of that region of the graph, omitting the data for cache sizes above 10000 Kbytes or below 5500 Kbytes. These graphs show that there is little effect from reducing the shrink fraction until the cache is quite small. In addition, the graphs show that the worsening performance for small shrink fractions is primarily due to increased garbage collection costs: there is little change in the fetching performance for the different shrink fractions.

Figure B-19 shows the effect of increasing the shrink fraction. Again, increasing the shrink fraction only affects the elapsed time noticeably for caches below 10000 Kbytes.

Figure B-20 shows the details of that region of the graph, omitting the data for cache sizes above 10000 Kbytes or below 5500 Kbytes. These graphs show that for shrink fractions of 0.7 and above, performance is consistently worse than for a shrink fraction of 0.5, and the cause of the worse performance is an increase in fetching that eliminates the gain from reduced garbage collection costs. A shrink fraction of 0.6 has performance approximately comparable to 0.5: neither consistently does better than the other.

Most subsequent variations of parameters give results similar to what has been described for TOTAL.

B.4.2 Varying the GC trigger

As has been the case for all policies, varying the GC trigger had little effect on the performance of the system.

Figure B-21 shows the effect of varying the gc trigger point when using the LRU policy.

B.4.3 Varying the Prefetch Algorithm

Figure B-22 shows the effect of varying the prefetch algorithm when using the LRU policy.

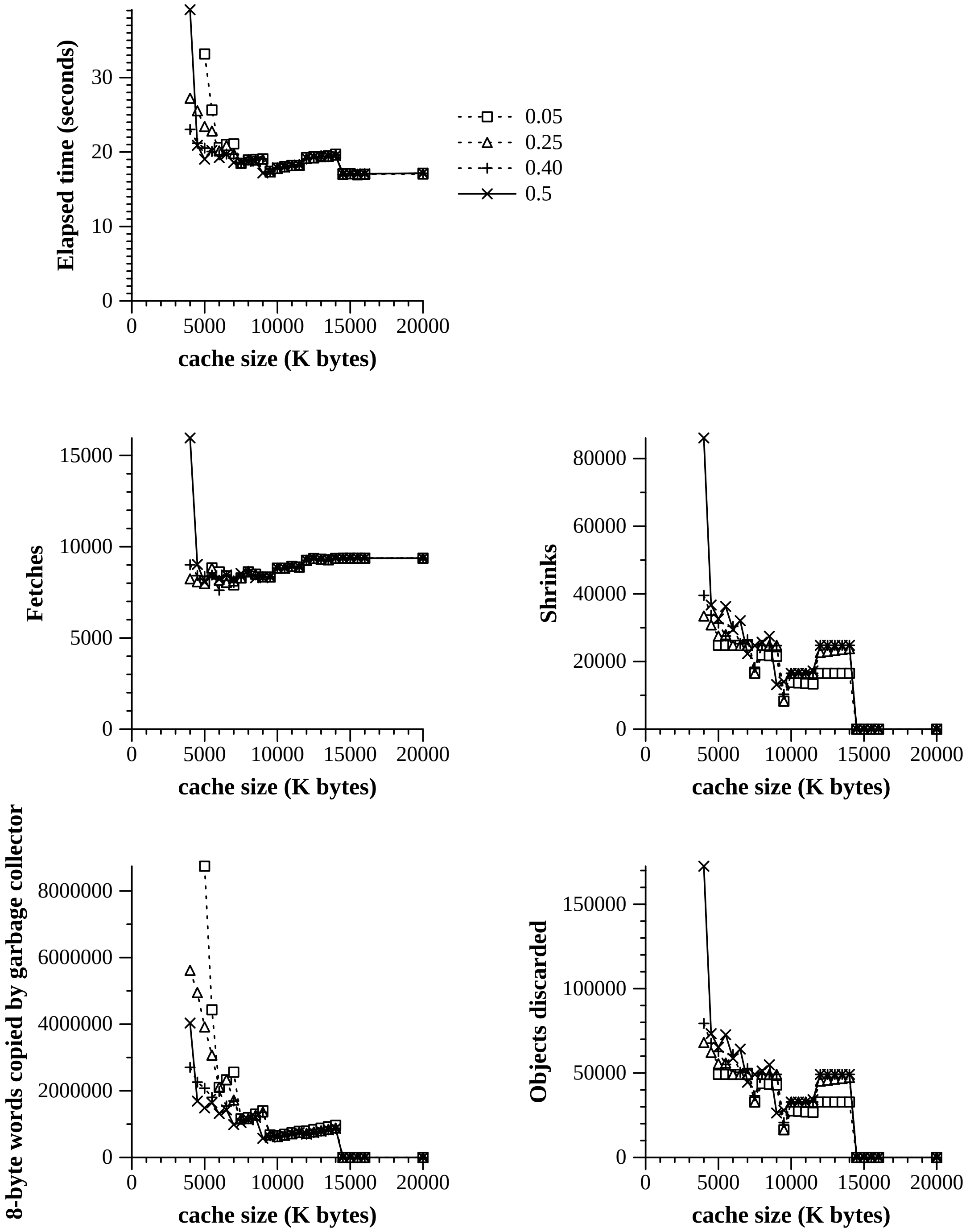

Figure B-17: LRU, reducing shrink fraction

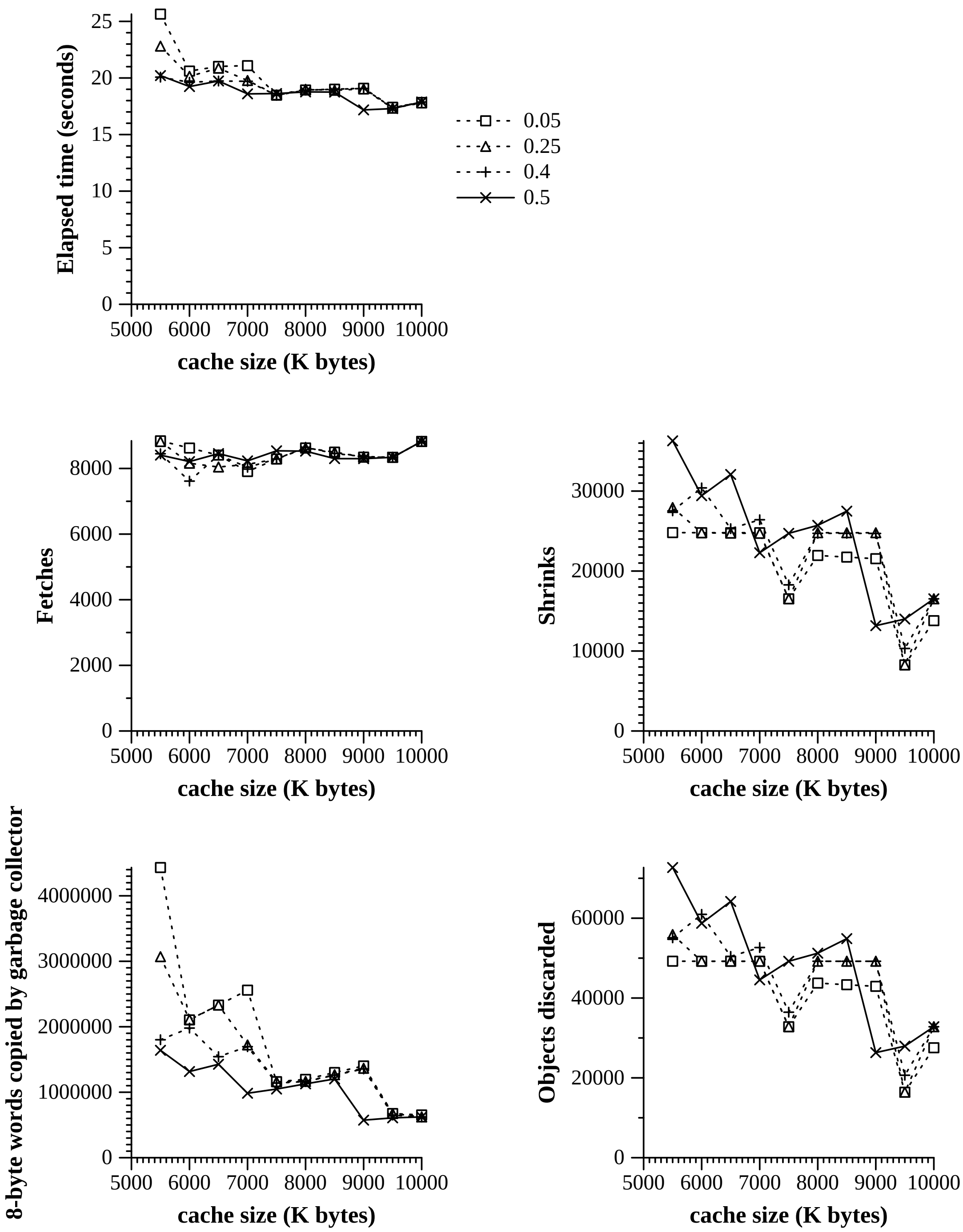

Figure B-18: LRU, detail of reducing shrink fraction

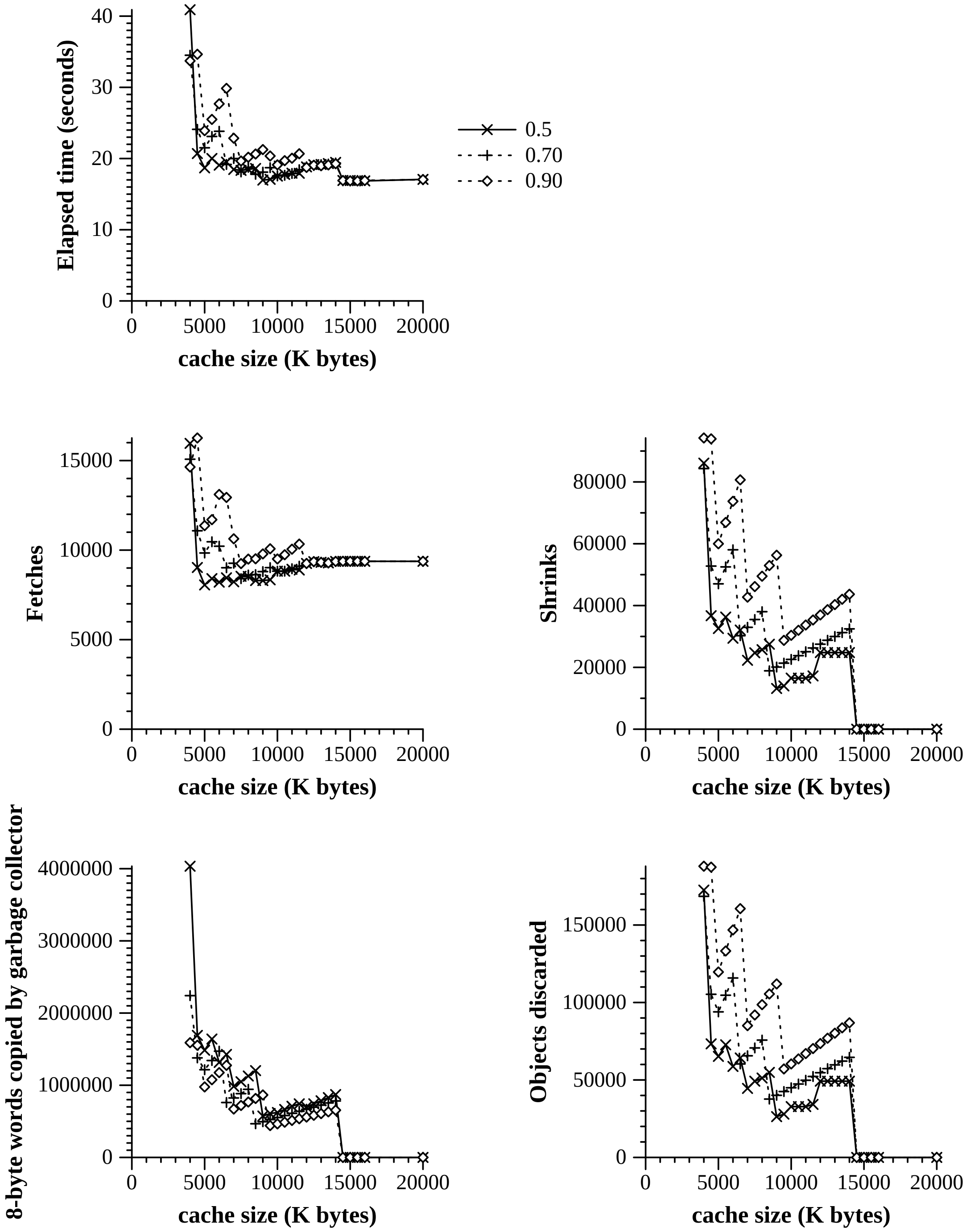

Figure B-19: LRU, increasing shrink fraction

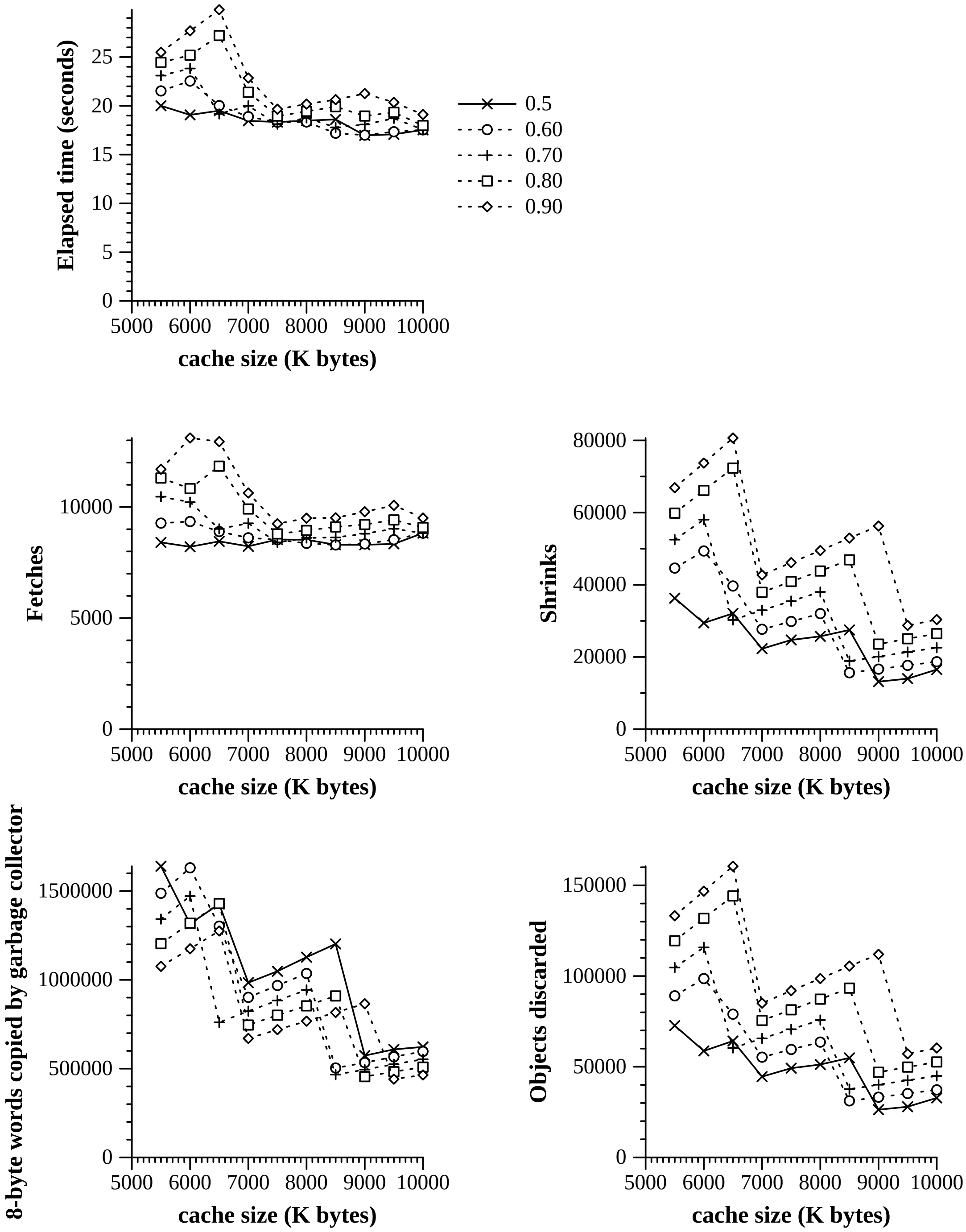

Figure B-20: LRU, detail of increasing shrink fraction

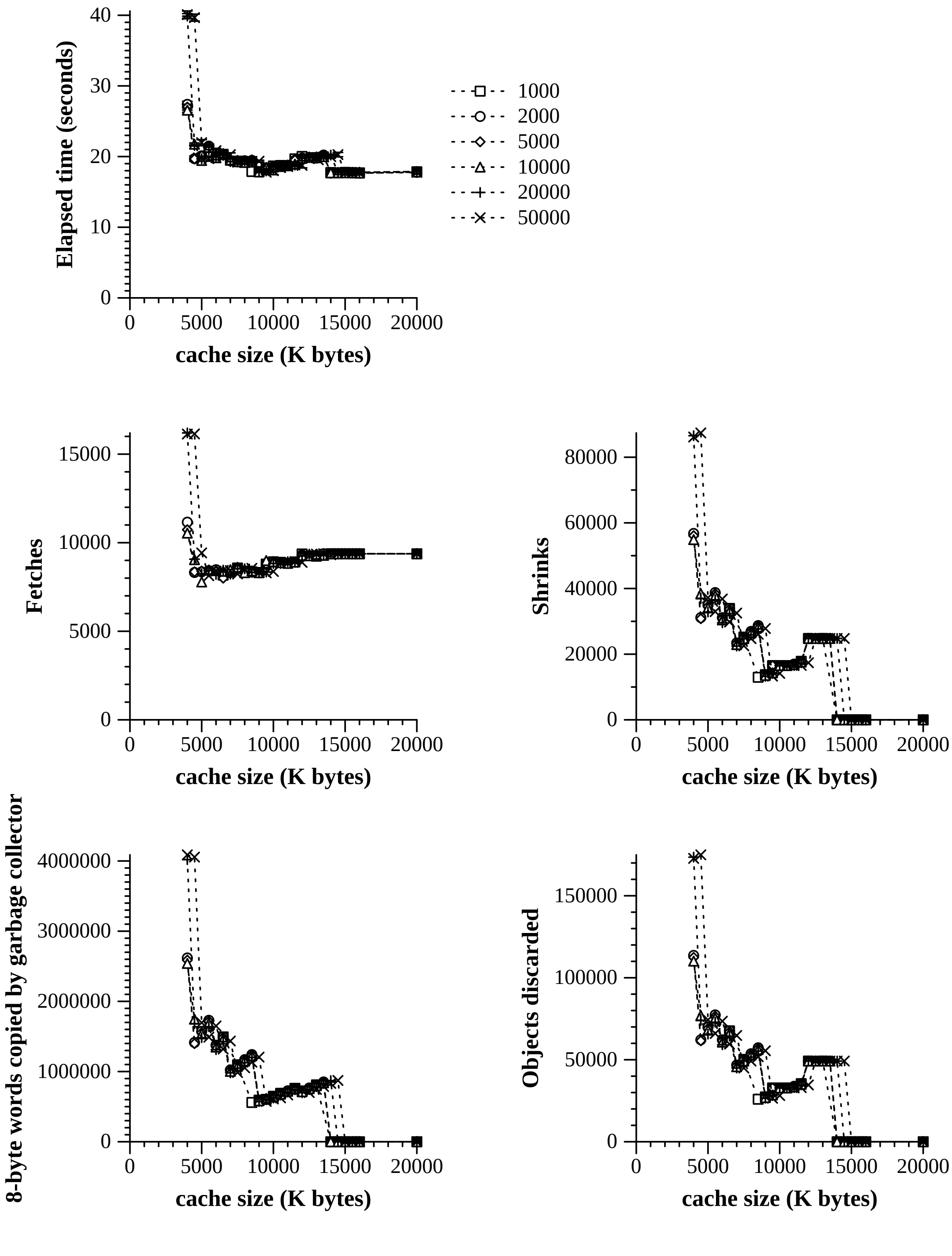

Figure B-21: LRU, varying gc trigger

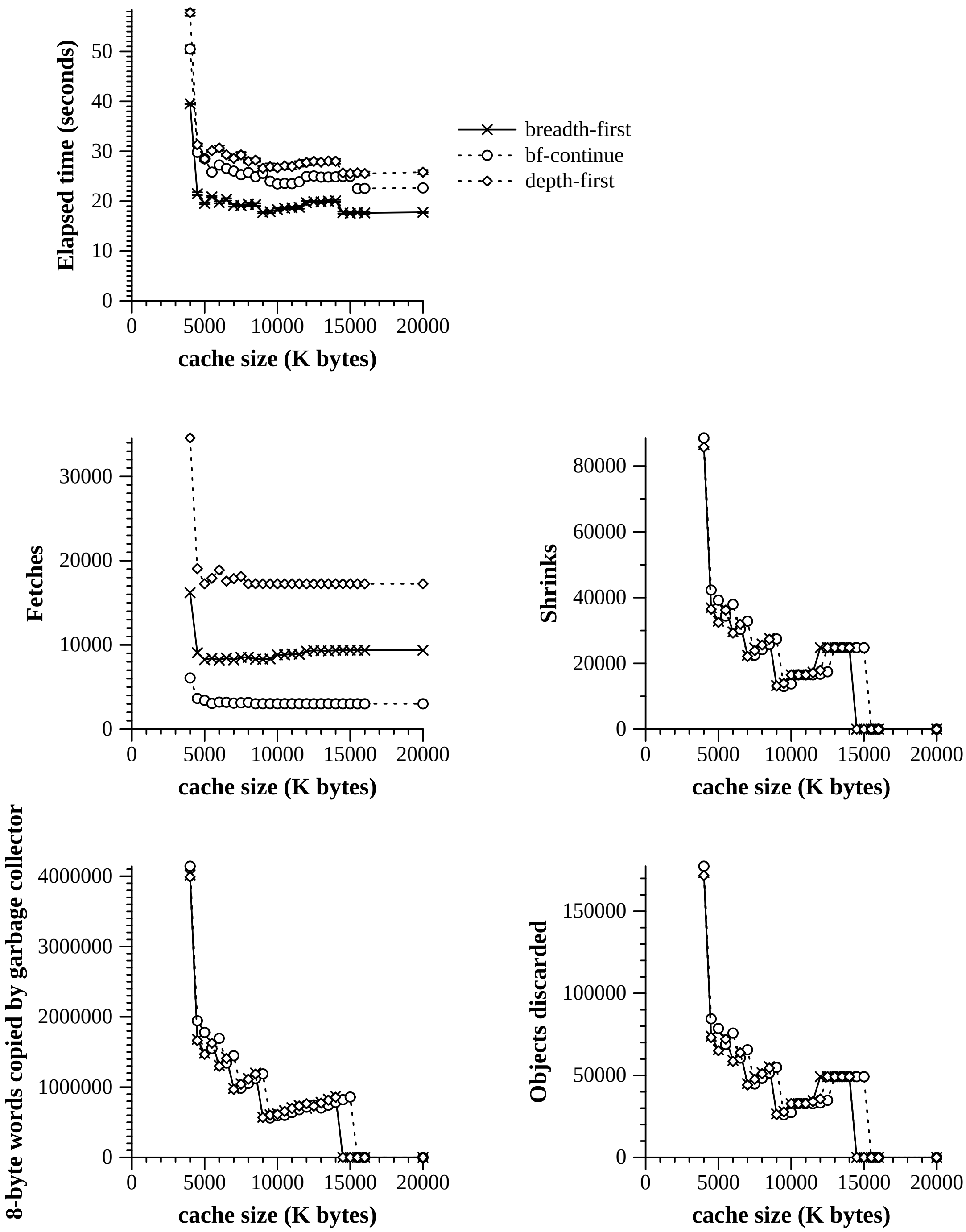

Figure B-22: LRU, varying prefetch algorithm

B.4.4 Varying the Maximum Prefetch Group Size

Figure B-23 shows the effect of decreasing the maximum prefetch group size when using the LRU policy. Figure B-24 shows the effect of increasing the maximum prefetch group size when using the LRU policy.

B.4.5 Varying the Shrink Trigger

Figure B-25 shows the effect of varying the shrink trigger when using the LRU policy. As has been true for other policies, a shrink trigger of 0.05 or less has much worse performance for smaller caches than other shrink triggers; and again as with other policies, the graph of gc copying (middle left) shows that for a shrink trigger of 0.05 there is too much garbage collection, repeatedly copying objects that should have been shrunk. Figure B-25 also shows that the shrink trigger only affects elapsed time significantly for cache sizes smaller than 10000 Kbytes.

Figure B-26 shows increased detail in the region of interest. There is substantial variation in the behavior of the system (how many objects are shrunk) but only small changes in the elapsed time. No value for the shrink trigger has a clear advantage over the default value.

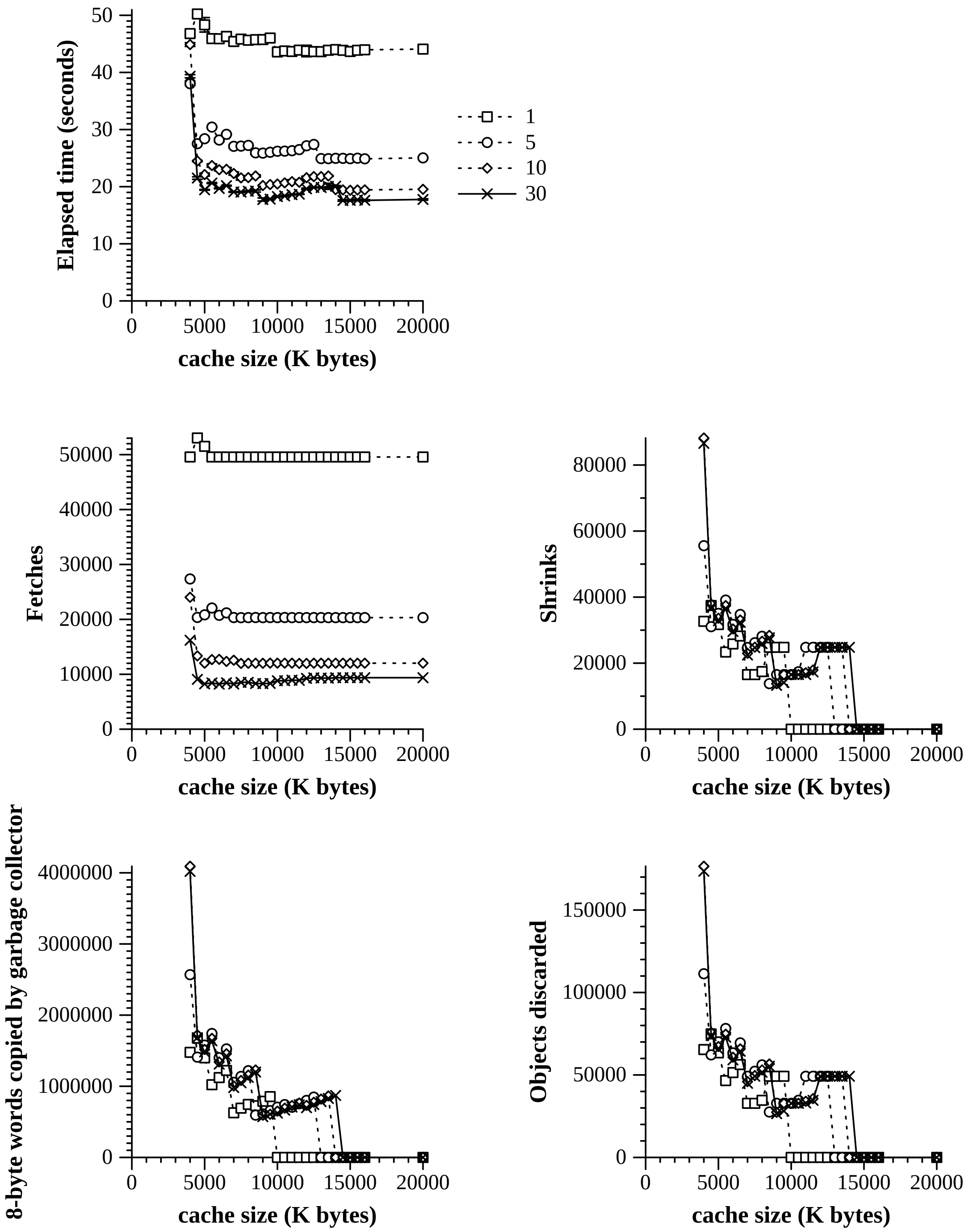

Figure B-23: LRU, decreasing maximum prefetch group size

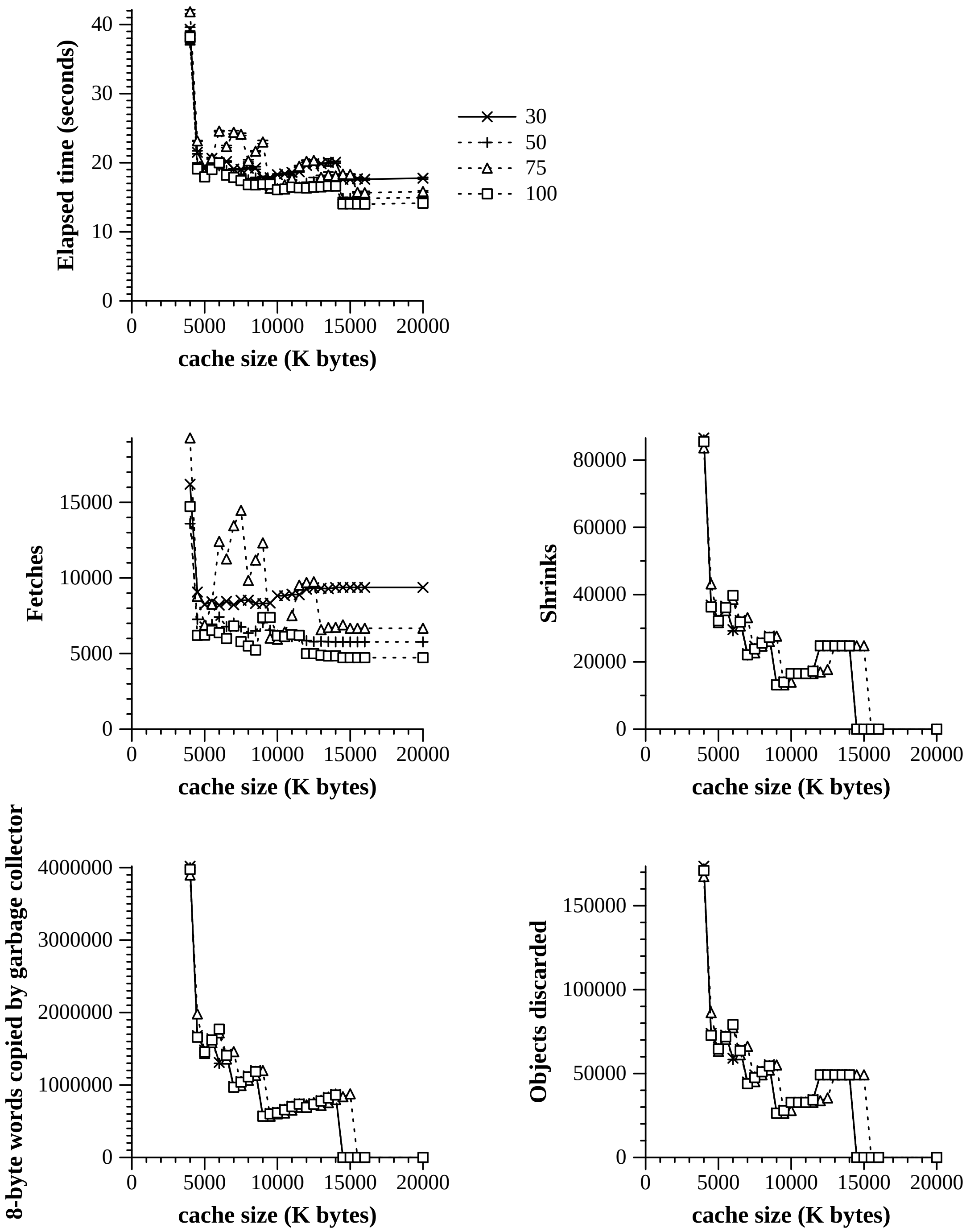

Figure B-24: LRU, increasing maximum prefetch group size

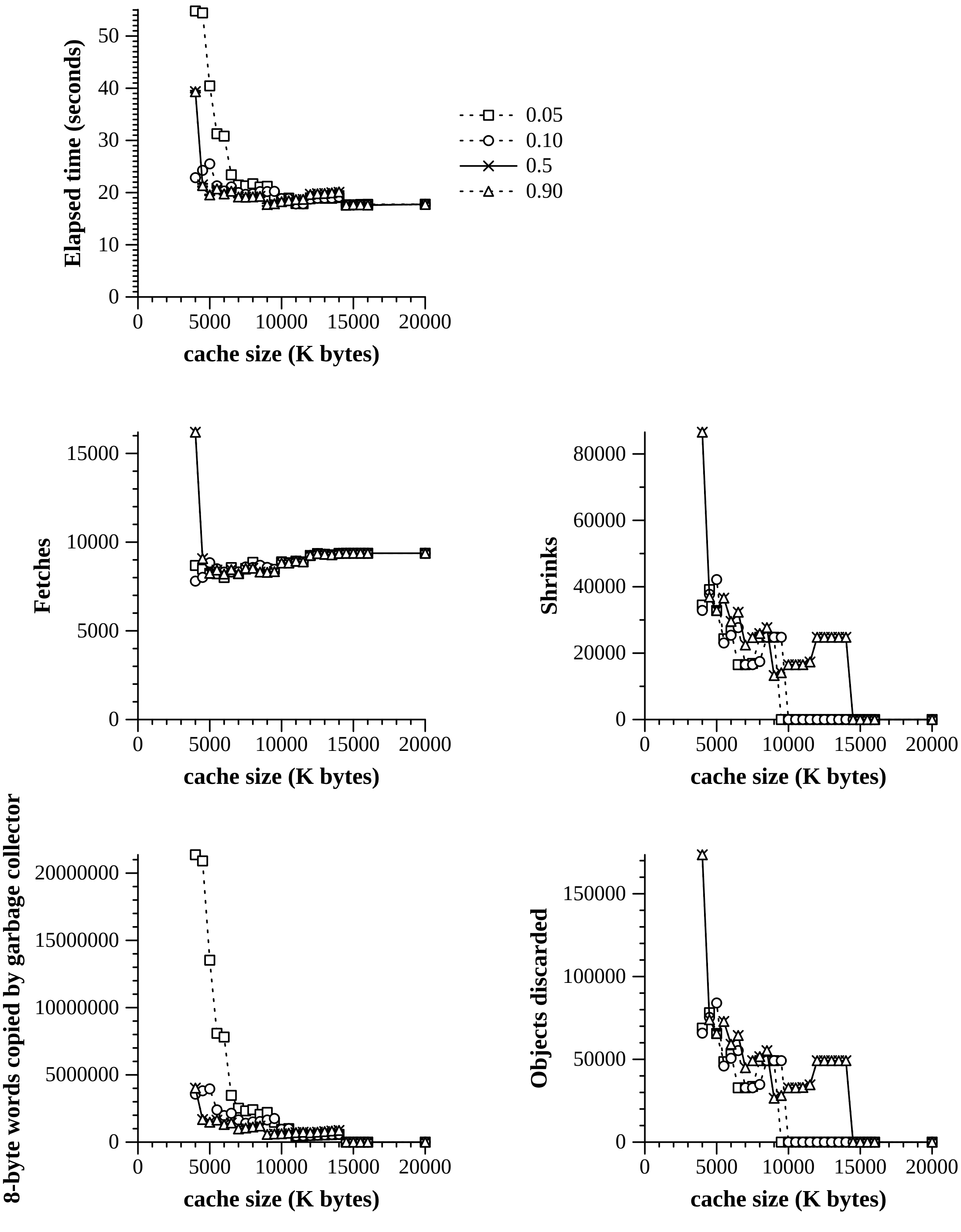

Figure B-25: LRU, varying shrink trigger

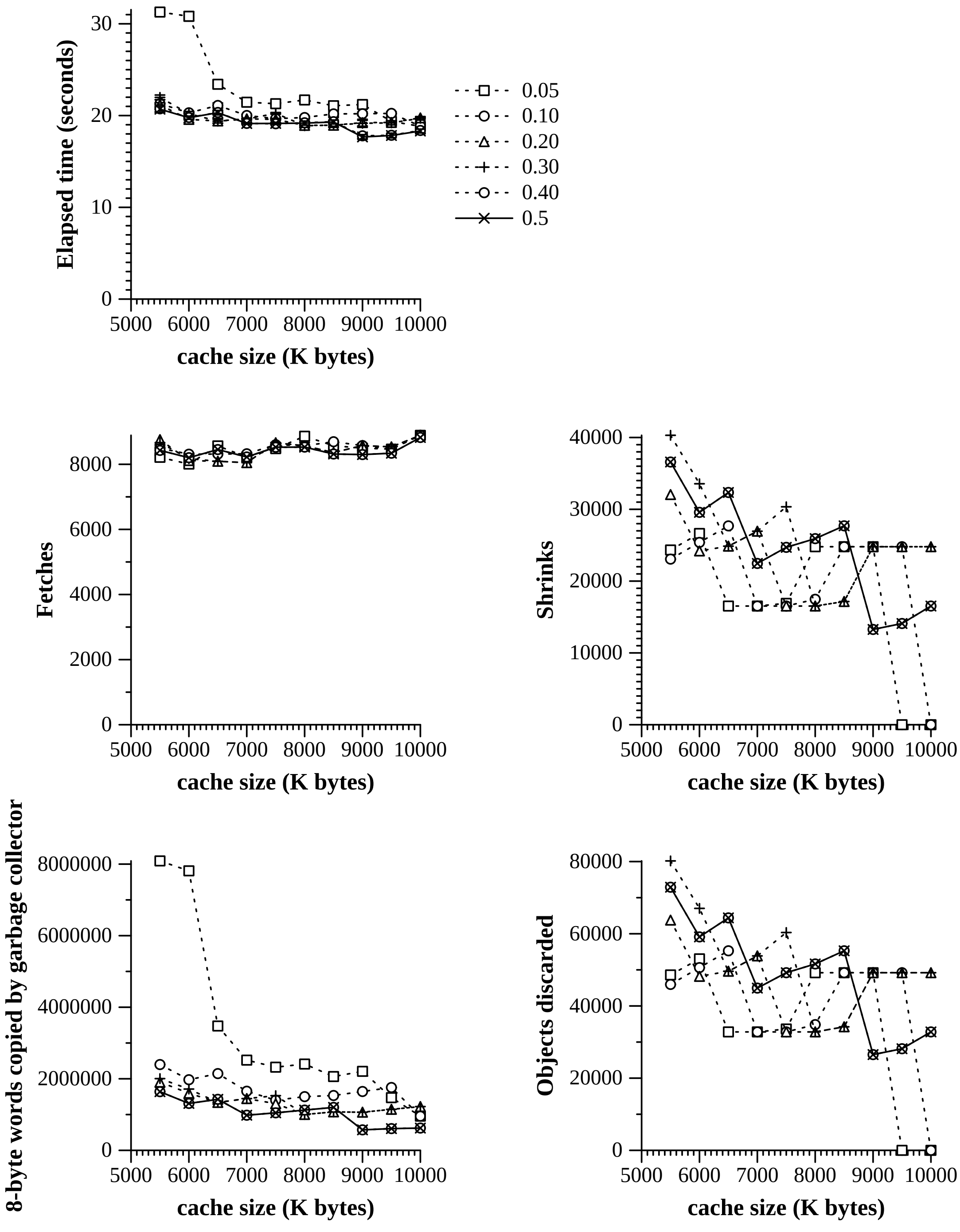

Figure B-26: LRU, varying shrink trigger