Optimizing Caching in Object-Oriented Applications

Kaushik Dutta*, Anindya Datta*, Helen Thomas†, Pinar Keskinocak‡

Abstract

Object-oriented (OO) technologies have become widely adopted in enterprise applications due to the additional functionality and flexibility they provide to these applications. At the same time, however, OO technologies also require significant amounts of computational power to support, greatly impacting the performance and scalability of such applications. A very popular solution to mitigate this problem is object caching. This technique has become so widely used that it is considered the defacto standard approach for optimizing application performance and scale.

In this paper, we show how the application of object caching maps into an optimization problem. In particular, we focus on the design-time decision of determining which objects should be candidates for caching. Choosing the cacheable objects is an important decision since it can have a significant impact on application performance. We formulate this problem as a linear integer program (IP) and present a heuristic solution approach. We show that our heuristic is efficient, having a running time complexity that is linear in the number of objects. We also demonstrate, through a set of experiments, that our heuristic provides solutions that are reasonably close to optimal. Our contribution is a model and an efficient solution approach for this model that can help application developers to make more informed cacheability decisions and thereby improve application performance and scalability. We believe that our proposed solution can be an integrated part of OO design products, such as Rational Rose and TogetherJ.

*Dupree College of Management, Georgia Institute of Technology, GA, Atlanta
†The Heinz School, Carnegie Mellon University, Pittsburgh, PA
‡School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA
1 Introduction

Object-oriented (OO) methodologies and technologies have received significant attention both in the academic literature and in practice during the last one and a half decades [10]. Initially, the market adoption of the OO paradigm lagged behind research, primarily due to the inertia induced by the deployment of large scale legacy information systems designed and built on non-OO methodologies. Such methodologies have included widely used logical and physical design paradigms such as the Entity-Relationship (E-R) and Relational models, as well as widely used implementation mechanisms like the FORTRAN and C languages. Since the mid-1990s however, there has been an explosion in the field of commercial deployment of OO systems – three distinct technological phenomena, encompassing the complete systems analysis and development spectrum from modeling to implementation, underscore this explosion.

1. There is virtually ubiquitous use of OO design techniques, the most popular of which is the Unified Modeling Language (UML) [12]. One would be hard pressed to find a single software house today that does not employ UML design products such as Rational Rose [24] or TogetherJ [25], in their systems design endeavors.

2. While Smalltalk [16], and eventually C++, signaled the advent of OO programming, two rival implementation methodologies have led to the true flowering of OO systems development, namely the Java programming language and associated development techniques and the Microsoft supported ASP/COM/.NET family of systems development methodologies. Any system developed using the above mentioned techniques are true OO systems, possessing many of the properties touted for fifteen years in the OO literature such as inheritance, strong and weak typing, and logic and data independence.

3. The widespread use of the World Wide Web (WWW) gave rise to an application environment where the technologies mentioned above could be applied, resulting in the deployment, on a massive scale, of OO applications that are used in day to day life. We invite the reader to consider the fact that whenever she visits a web site built on Java or ASP/COM technologies (to browse an online bookstore or to check airplane fares, for instance), she is interacting with
a true OO application system. Given that the Java 2 Enterprise Edition (J2EE) [20] and the
ASP/COM architectures [7] (soon to be .NET [8]) are the most popular dynamic web site design
architectures today, it would not be overreaching to claim that OO technologies touch our lives
everyday in multiple ways.

Unfortunately, while adding tremendous amounts of functionality and flexibility to applications,
OO technologies come with a significant drawback. The very features which underscore the
power of the OO paradigm, also require significant amounts of computational power to support,
greatly impacting the performance and scalability of such applications. For example, enterprise
OO applications are characterized by the continuous creation, consumption, and destruction of
various types of application objects, e.g., product objects (in e-commerce applications), session
objects, and user profile objects. Creation and destruction of these objects is computationally
very expensive: object creation usually requires accessing persistent storage in back-end systems
(e.g., DBMSs and file systems) and subsequent processing of this information; object destruc-
tion necessitates frequent invocation of CPU-intensive memory management or garbage collection
processes [21].

A very popular solution to mitigate this problem is object caching. Here, an object generated
for a particular request is saved (usually in application memory) and used to serve subsequent
requests for the same object. Nearly all enterprise software systems (whether coded from scratch
or running packaged applications) support object caching. J2EE application servers (e.g., BEA’s
WebLogic [26], IBM’s WebSphere [13]) and packaged applications (e.g., Siebel [27], SAP [23]) are
common examples of such enterprise software solutions that offer object caching features. In fact,
object caching has become so widely used that it is considered the defacto standard approach
for optimizing application performance and scale. JCache [22], a proposal for a standard Java
API for object caching, is evidence of this.

To effectively integrate object caching into an application infrastructure, two important deci-
sions need to be made. The first decision is to determine, at design time, which object types
should be candidates for caching. The second decision is to determine, at run-time, which specific object
instances should be cached. The latter problem maps into the well-known cache replacement
problem in operating systems theory and can, consequently, use the existing solutions.

The former problem is the focus of this paper. Choosing the cacheable objects\(^1\) is an important decision since it can have a significant impact on application performance. An incorrect cacheability decision made at design time can have debilitating effects on the run-time performance and scalability of applications - in fact, it may nullify the benefits of caching completely. Interestingly, the cacheability decision is currently made in a very ad hoc manner, mainly because there are no systematic processes nor any commercially available tools to help identify the appropriate objects to be cached. Thus, the selection of cacheable objects is left to the discretion of the application developer. This may (and often does) result in sub-optimal decisions, which can degrade run-time application performance. For example, if the run-time cost to fetch an object from cache is on the same order as the run-time cost to generate the object, then there is no benefit gained from caching the object. Rather, there is additional overhead since the run-time cache manager must consider the object when enforcing caching policies.

It would not be overreaching to claim that poor identification of cacheable objects is perhaps the leading cause of unexpected performance and scale problems in enterprise applications. We discuss this cacheability decision problem and demonstrate that it can be modeled as an optimization over a number of systems parameters. We formulate this problem as a linear integer program (IP) and present a heuristic solution approach. We show that our heuristic is efficient, having a running time complexity that is linear in the number of objects. We demonstrate, through a set of experiments, that our heuristic provides solutions that are reasonably close to optimal. Our contribution in this paper is a model and an efficient solution approach for this model that can help application developers to make more informed cacheability decisions and thereby improve application performance and scalability. We believe that this solution can be an integrated part of OO design products, such as Rational Rose [24] and TogetherJ [25]. As far as we know, this is the first attempt to address the critical problem of choosing the appropriate set of cacheable application objects.

The remainder of the paper is organized as follows. In section 2 we explain the root causes

---

\(^1\)Throughout this paper, we use the term *object* to refer to an object type, and *instance* to refer to a specific instantiation of an object type.
behind the performance and scalability problems native to OO systems and motivate the need
to employ object caching as a solution. In section 3 we present an IP formulation to model
the decision problem of optimally choosing cacheable objects. In section 4 we discuss solution
approaches and present a heuristic algorithm to solve the problem. In section 5 we present
experimental results to measure the performance of the heuristic, and in section 6 we discuss
related work. We conclude in section 7.

2 Problem Motivation

To better understand the performance and scalability issues associated with OO technologies,
we discuss the inner workings of OO information systems. In general, any information system
is composed of data, which is usually stored in persistent storage (e.g., hard disk), and a set
of applications that operate on the data. In object-oriented systems, each real life entity is
represented by an object, and the application logic is defined as an interaction between a set of
objects. Such logic is encapsulated in the behavior specifications of these objects.

We can thus represent any OO application as a set of complex objects \( \{o_1, o_2, \ldots, o_n\} \), where
each object \( o_i \) consists of an object hierarchy. Each object may be composed of primitive data
types (i.e., the lowest-level undecomposable objects) or other objects. Higher level objects in the
hierarchy are related to lower level objects by a “contains” relationship. For example, consider a
simple banking system which consists of the objects customer, account, and bank. Figure 1 shows
the object hierarchies for the account and customer objects. An account object consists of account
ID, type, and balance objects, and a customer object consists of personal information and offers
objects. The account object contains sub-objects account ID, type, and balance. Account ID and
type are both primitive data types, whereas the balance object is a complex object, containing
primitive data types, as shown in the figure.

When an OO application is designed, object templates are created to specify the structure and
behavior of each object. When a request is made to an OO application at run-time, the object
templates are used (along with data stored separately) to create instances of the objects. These
object instances are then manipulated as necessary by running the application or business logic.
Finally, after the application logic has executed, the instances are destroyed. *As it turns out, a significant portion of the work required to serve requests in OO applications is in the creation and destruction of object instances.* We now discuss the object instance creation and destruction processes in more detail, and present two detailed examples illustrating these processes.

Instance creation occurs at run-time when a request is made by fetching the appropriate object class template and data and subsequently instantiating the template. The instance creation process can incur significant delays for several reasons: (i) Accessing the template and data from persistent storage (often a relational database management system) is slow due to the physical I/O required, (ii) Accessing data from databases requires the use of database connectivity software (e.g., ODBC/JDBC), which adds overhead, (iii) With the increasing popularity of hosting facilities, different pieces of data required to compose an object instance may reside at different locations, further adding to the delay in creating object instances, (iv) Applications typically have multiple layers, and thus a given request may invoke logic that is nested several levels deep, and (v) Instantiating a complex object involves instantiation of all its constituent parts.

To make matters worse, once all the logic has run to satisfy the request, the instances must be destroyed, which involves certain cleanup operations. For example, the instance may be saved to persistent storage, which may involve any or all of the above-mentioned delays associated with data access.

Another issue associated with instance creation and destruction is memory management. Each instantiation of an object consumes a certain amount of memory. Thus, when an instance is destroyed, this memory must be freed. The repeated creation and destruction of object instances
has a significant impact on application performance due to the additional overhead of memory management that is required. Such memory management is commonly referred to as *garbage collection*.

We next present two examples, a banking application and a web-based office supply application, to illustrate the performance and scalability issues associated with OO applications. We also demonstrate how caching is used to ameliorate these problems.

### 2.1 Banking Example

Consider again the banking example (refer to Figure 1) supporting two types of transactions, deposits and withdrawals. Examples of business logic in this case would be making a withdrawal or making a deposit, both of which involve the *customer* and *account* objects. Assume that the object class templates as well as application data are stored in persistent storage.

Suppose a customer, John, makes a request to withdraw $100 from his checking account. To respond to this request, the OO banking system accesses the *customer* and *account* object templates and the appropriate data required for object instantiation, and creates instances of the “John” *customer* object and the “account 401” *account* object. Needless to say, the *account* and *customer* instances for “John” and “account 401” require that the nested object hierarchies for the account and the customer object shown in Figure 1 be created. Suppose that the *balance* object is instantiated by accessing a local database and retrieving John’s current balance. Furthermore, the *personal information* object instance is created by accessing a database in a remote data center where secure customer information is stored, and the *offers* object instance is created by accessing a local database which contains information regarding special offers available for preferred customers. Instantiation of the object hierarchy in this simple example requires significant work, involving both physical I/O and network delays.

After the necessary objects have been instantiated, the appropriate logic executes. First, the logic that computes John’s new balance runs. Assuming that John has a sufficient balance to cover the withdrawal, this logic will update John’s *account* object instance accordingly. Then, perhaps additional logic runs that checks to see whether John qualifies for any special offers.
After all logic has executed, the object instance destruction process takes place. Since John’s account balance was updated, the balance object instance must be written to the local database. If John qualified for a special offer, then the offer object instance is also written to the appropriate database. Each of the instances are then destroyed.

Without the overhead of instance creation and destruction, the performance of this application would be much better in a non-OO system than in its OO counterpart. In real-life applications, the logic is typically significantly more complex than the example above and it is often the case that hundreds or thousands of requests must be served simultaneously. The end result is serious performance and scalability problems for OO applications.

2.1.1 Object Caching

A technique to remedy the above-mentioned delays is object caching, which involves storing instantiated objects in memory to serve future requests rather than destroying them. By keeping object instances in local memory, subsequent requests for the object instances can be satisfied directly from memory, reducing the overhead of instance creation. For example, suppose John returns to the banking application to review his balance shortly after making his withdrawal. If the “John” instance of the customer object and “account 401” instance of the account object are cached, then the overhead of creating these instances is eliminated.

2.2 Web-Based Example

In this section, we illustrate the performance and scalability problem of OO systems with a World Wide Web application example. Modern dynamic web sites generate pages on the fly. The applications that generate these dynamic pages are classic examples of OO systems and are built using the Model-View-Controller (MVC) paradigm [3]. In this paradigm, the user deliverable web page object is composed of a set of presentation layer objects. Each presentation layer object is in turn composed of a set of application layer objects. In MVC, the view layer refers to the presentation layer, the model layer refers to the underlying data layer, and the controller layer refers to the interaction or application logic layer. Thus, each request for a page
in an MVC-based site requires the creation of object instances and the interaction among these instances at multiple, nested layers. We now present an example in order to describe this problem in more detail.

Consider an online office supply web site. Visitors to the site have the option to create a login profile so that they can receive personalized offers. A typical interaction with the site proceeds as follows. A user logs on to the site and is uniquely identified. The user’s identification is used to access the user’s profile from the database. The user’s profile is built as the user makes purchases from the site, and is represented by a set of keywords that denotes which category/subcategory/product the user had interest in during his/her previous visit. Based on these keywords, the appropriate offers are selected for the user.

Suppose a visitor to the site, Bob, has entered his login information (user ID and password) through his web browser. This request invokes a page generating application at the office supply site, which generates the page that is returned to Bob. A mockup of the resulting web page is shown in Figure 2[A]. This page can be thought of as an object instance, which is itself composed of several presentation-layer object instances: Top Nav Bar, Right Nav Bar, Left Nav Bar, Middle Content, and Bottom Nav Bar. These objects in turn may be composed of application-layer object instances. For example, the Left Nav Bar object instance is composed of General Info and Category. The Category object instance is further composed of the Sub-Category object instance. The corresponding object hierarchy is depicted in Figure 2[B]. Although not shown in the figure, an object may be contained in multiple objects, a concept referred to as multiple inheritance. However, multiple inheritance is not used often in practice - in fact, it is not supported in Java.

Bob’s initial request which generated this page required the instantiation of every object shown in Figure 2[B], along with the execution of the appropriate business logic. Suppose another visitor, Alice, enters the office supply site shortly after Bob. The resulting page that is generated for Alice’s request is nearly identical to the page that was created for Bob, except that the offers instances, Free Offer, Hot Deal, Great Deal, and Special Offers, will be different. In the absence of object caching, every instance will have to be created again, even though only
a few instances actually changed. This is a classic problem faced by virtually all dynamic web sites.

### 2.2.1 Object Caching in Web-Based Applications

If object caching were employed, many of these instances, once created, could be stored in memory to serve subsequent requests, eliminating much of the overhead associated with instance creation and destruction. This is precisely the idea behind *dynamic content caching* solutions. Some of these solutions provide *fragment-level* caching. In this approach, each page is represented as a set of fragments, and the fragments deemed to be reusable are cached. A fragment is simply an object instance (e.g., presentation or application-layer object instance). Referring back to our example, these solutions may cache presentation-layer instances, i.e., HTML fragments, such as the **Top Nav Bar** or **Middle Content**. In addition, some of these solutions are capable of caching application-layer instances, such as **Sub-Category** or **Related Product** instances.

Some of the commercial dynamic content caching products which offer fragment-level caching include EdgeSuite from Akamai [28] and Oracle 9i Web Cache [6]. Many application servers, such as WebLogic from BEA Systems [26] and WebSphere from IBM [13], also provide fragment-level caching.
2.3 Problem Statement

Given an object hierarchy, one can think of the caching problem at two levels: the design level and the run-time level. At the design level, the choice to be made is to determine which of the objects in the hierarchy should be made “cacheable”. For example, in our web page example in Figure 2[B], it may be the case that only the Left Nav Bar and Middle Content objects should be “marked” as cacheable. Once an object, say Left Nav Bar, is “marked” cacheable at design time, any created instance of the Left Nav Bar object would be a candidate for caching by the run-time cache manager.

At run-time, the cache manager decides, on the fly, whether to cache a specific object instance. Also, if the cache manager decides to insert a specific object instance into cache and the cache happens to be full, an additional decision that must be made is what to discard from cache to make room for the newly arrived instance. This is the well-known cache replacement problem, which has been studied extensively in the operating system literature (e.g., [19, 31, 4]).

Our focus in this paper is on the design level problem. One may wonder why we would not simply mark all objects as cacheable (i.e., any object at any level in every hierarchy). Under this scenario, all caching decision making would be offloaded to the run-time cache manager as every instance of every object created would be a candidate for caching. As it turns out, this strategy degrades the performance of the run-time cache manager in several ways. For every requested object, there is a run-time cost to fetch the instance from cache, referred to as the cache fetch cost. Each object also has a generation cost, which is the cost to create the instance. If the cache fetch cost and generation cost are of the same order for an object, then it is not beneficial to cache the instance. For example, in our example application, if it takes about 30 ms on average to generate the Left Nav Bar instance and about 50 ms to retrieve the instance from cache, then it is not worth caching this instance. Rather, it would be worthwhile to explicitly mark this object as “not cacheable” at design time, thereby freeing the run-time cache manager from considering it.

Another issue is that some instances may not remain in cache long enough to serve subsequent requests. In other words, if the update rate of an instance is greater than the request rate, then
it is not beneficial to cache the instance. For example, if the **Personalized Content** object in our example contains the current date and time, then each time an instance of this object is placed in cache, it immediately becomes invalid. Thus, the decision to cache this object adds overhead, which can degrade system performance. On the other hand, if the **Personalized Content** object contains information that is updated on average every 5 minutes and there are multiple requests during this period, then it is beneficial to cache instances associated with this object. In summary, marking every object as cacheable increases the overhead of the run-time cache manager by increasing its decision space exponentially.

For the reasons discussed above, choosing the set of cacheable objects in advance can significantly improve system performance, and these decisions are the focus of this paper.

## 3 The Model

In this section we develop a cost/benefit model for caching that can be used to decide, at design time, which objects are the best candidates for caching. Application of our methodology on a set of objects will yield “marked” objects - those objects that are deemed to be the best candidates for caching. To model the caching problem, we first define the structure of the objects. We then present the decision variables, objective, parameters, constraints, and finally, the complete formulation of the model. Table 5 in the Appendix contains the notation that is used throughout this section. The notation will be described in the text as well.

### 3.1 Object Structure

An important aspect of our cost/benefit model is the structure of the underlying objects and their relationships to one another. Using the notion of an **object hierarchy** (discussed in Section 2), we model an application object as a set of nested objects which can be represented by a directed, acyclic graph \( G = (V, E) \), where \( V \) is the set of nodes and \( E \) is the set of directed edges. Each node in the graph represents an object. For two nodes \( u \) and \( v \), a directed edge \( (u, v) \) exists if object \( u \) is **contained by** object \( v \). For example, in Figure 2[B], the **Right Nav Bar** object contains the **Advertisement**, **Special Offer**, and **Articles** objects, as indicated by the directed
arcs. In order to generate a **Right Nav Bar** object instance, instances must be generated for these three objects.

Given the graph $G$, we distinguish between two types of objects: (i) *root objects* and (ii) *non-root objects*. Root objects are those that appear at the root of object hierarchies - they are application level objects that exist independently within the system. Non-root objects are objects that are contained by a root level object or another non-root level object. In Figure 2[B], **Homepage** is a root object that contains five non-root objects: **Top Nav Bar, Right Nav Bar, Left Nav Bar, Middle Content** and **Bottom Nav Bar**. Each of these objects in turn contain other non-root objects and so on.

We now define additional notation that is used throughout the remainder of the paper. Let $I$ be the set of all objects and let $i \in I$ be an individual object. For each object $i$, we define $D(i)$ to be the set of objects that are related to object $i$ through the contained by relationship. In Figure 2, $D(\text{Left Nav Bar}) = \{\text{General Info, Category}\}$. Let $J$ be the set of *deliverable objects* such that $J \subseteq I$. An object $j$ is *deliverable* if it is accessible by an end user. Objects that are not deliverable can only be accessed by higher level objects. Referring again to Figure 2[B], **Homepage** is a deliverable object since it can be accessed by an end user. Root objects are by definition deliverable objects; however, deliverable objects are not necessarily root objects. For instance, if the **Middle Content** object in Figure 2[B] can be accessed by an end user (e.g., through a URL), then it is a deliverable object. We define $\text{subtree}(j)$ to be the set of all nodes in $I$ which are directly or indirectly contained by object $j \in J$. For example, referring again to Figure 2[B], $\text{subtree}(\text{Left Nav Bar}) = \{\text{General Info, Category, Sub-Category}\}$.

We let $S_i$ denote the size of an object $i$, which represents the average size of an instance of the object. $S_i$ is given by $S_i = S_{si} + \sum_{j \in d(i)} S_j, \forall i \in I$, where the first term represents the intrinsic size of the object and the second term represents the total size of the object instances that are nested within the object’s hierarchy.
3.2 Decision Variables

The decisions in the model are which objects to mark as cacheable. We represent these decisions using binary variables $X_i$, $\forall i \in I$, such that $X_i = 1$ if an object is marked as cacheable and $X_i = 0$ otherwise.

3.3 Objective Function

The objective of the model is to identify the objects to be marked as cacheable so that the total cost of accessing all root objects is minimized. For a given root object, this total cost represents the cost to create an instance (and all of its contained object instances) less the benefit obtained from marking the object cacheable. Expressions for these costs and benefits are derived next.

3.3.1 Generation Cost

The generation cost for an object refers to the cost to create the object, which is the cost to perform the tasks described in Section 2 (e.g., accessing the object template, accessing the application data, instantiating all nested objects, and destroying the object instances). Within our framework, we classify the object generation process and related costs into two types of tasks: (a) generation of the object instance itself, which incurs intrinsic generation cost, $I_{gi}$, and (b) generation of the object instances that are nested within the object’s hierarchy, which incurs nested object generation cost. For example, for the Bottom Nav Bar object in Figure 2, the intrinsic generation cost is the cost to create the Bottom Nav Bar object instance, while the nested object generation cost is the cost to create the nested Static Nav Bar and Personalized Content object instances.

We define $C_{gi}$ to be the cost to generate an instance of object $i$, and $C_{fi}$ to be the cost to fetch an instance of object $i$ from cache. In the case of an object that is not marked, the cost to access the object instance for a given request will be $C_{gi}$. In the case of a marked object, the cost to access the object instance for a given request will be $C_{gi}$ if the instance is not found in cache, and $C_{fi}$ otherwise. We define the hit ratio to be the percentage of total object instance requests that are served from the cache. Since the cache replacement policy is applied globally,
i.e., the replacement policy is applied to all cached object instances, we define $h$ to be the global hit ratio for the cache.

We can now derive an expression for the average access cost over all object instances. The average access cost of an object that is not marked is simply $C_{gi}$. For marked objects, the average access cost depends upon the hit ratio and is given by $(hC_{fi} + (1 - h)C_{gi})$. Hence, the total generation cost for an object $i$ is given by:

$$C_{gi} = I_{gi} + \sum_{j \in d(i)}[X_j\{hC_{fj} + (1 - h)C_{gj}\} + (1 - X_j)C_{gj}], \forall i \in I \ (1)$$

In (1), the first term represents the intrinsic generation cost, and the second term represents the nested object generation cost (as indicated by the summation over the set of contained objects). An important observation is that (1) is recursive, which leads to a nonlinear expression in the variable $X_i$. Nonlinear integer programs are typically complex and very difficult to solve. Therefore, we follow an alternative approach and model this problem as a linear integer program (IP).

To derive a linear expression for generation cost, we consider the generation cost for a root object $i \in J$ to be composed of three costs: (i) the fetch cost of all cacheable objects in the subtree rooted at $i$, (ii) the intrinsic generation cost of all cacheable objects in the subtree rooted at $i$, and (iii) the intrinsic generation cost of all non-cacheable objects in the subtree rooted at $i$.

Consider the simple example object hierarchy shown in Figure 3, where $C$ denotes an object marked as cacheable, and $NC$ denotes an object that is not marked as cacheable.

![Figure 3: Example Object Hierarchy](image)

In computing the total generation cost for the root object, object 1, we need to determine the fraction of accesses for which a particular object instance is generated and the fraction of accesses for which an object instance is fetched from cache. If object instance $i$ is in the cache, then we do
not need to consider the generation cost of the objects in the subtree rooted at \( i \). Alternatively, let \( \text{path}(i) \) denote the unique path from object \( i \) to the root. If any \( j \in \text{path}(i) \) is in the cache, then we do not need to consider the generation cost of object instance \( i \). Since a cacheable object instance is not in the cache \((1 - h)\) fraction of the time on average, the fraction of time that one of the \( n \) cacheable object instances in \( \text{path}(i) \) is not in cache is \((1 - h)^n\). Hence, if there are \( n \) cacheable objects on \( \text{path}(i) \), we need to consider the generation cost of object \( i \) \((1 - h)^n\) fraction of the time.

Referring back to our example in Figure 3, we first consider the left subtree. Note that we only need to consider the generation cost for object 2 when the instance corresponding to object 1 is not found in cache, i.e., for the \((1 - h)\) fraction of accesses for object 1 that are cache misses. Since object 2 is cacheable, we must consider whether the instance associated with object 2 is generated or fetched from cache. For the \((1 - h)\) fraction of accesses for this object that are misses, object instance 2 will have to be generated, and for the remaining \( h \) fraction of accesses, object instance 2 will be fetched from cache. This gives us \((1 - h)^2\) as the fraction of accesses for object instance 2 which will have to be generated, and \(h(1 - h)\) as the fraction of accesses for object instance 2 which will be fetched from cache.

Similarly, the results for the other objects are summarized in Table 1. Note that object 3 does not have an entry for the fraction of fetch accesses since it is not marked as cacheable, and hence will never be fetched from cache.

<table>
<thead>
<tr>
<th>Object</th>
<th>Fraction of Fetch Accesses</th>
<th>Fraction of Generation Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>( h(1 - h) )</td>
<td>((1 - h)^2)</td>
</tr>
<tr>
<td>4</td>
<td>( h(1 - h)^2 )</td>
<td>((1 - h)^3)</td>
</tr>
<tr>
<td>3</td>
<td>( n/a )</td>
<td>((1 - h))</td>
</tr>
</tbody>
</table>

Table 1: Example Generation Cost Computation

To be able to express the generation cost as a linear function, we define the variables shown in Table 2.

The fetch cost of all cacheable objects in the subtree rooted at \( i \) can be expressed as \( \sum_{j \in \text{subtree}(i)} \sum_{k=1}^{d_{ij}} Z_{ijk}(1 - h)^{(k-1)}hC_{ij}. \) Since \( \sum_{k=1}^{d_{ij}} Z_{ijk} = 1 \), only one of the terms in the inner summation will be positive, determining the number of cacheable objects on the path from \( j \) to \( i \). The intrinsic generation cost of all cacheable objects in the subtree rooted at \( i \) is given
\[ Y_{ijk} = \begin{cases} 0, & \text{if the number of cacheable objects on the path from object } j \text{ to } i, \text{ including } i, \text{ is } k; \\ 1 \text{ otherwise}, \end{cases} \]

\[ Z_{ijk} = \begin{cases} 1, & \text{if object } j \text{ is cacheable and there are } k \text{ cacheable objects on the path from object } i \text{ to } j, \text{ including } j; \\ 0 \text{ otherwise}. \end{cases} \]

Note that \( \sum_{k=1}^{d_{ij}} Z_{ijk} = 1 \), where \( d_{ij} \) is the length of the unique path from \( j \) to \( i \).

\[ W_{ijk} = \begin{cases} 1, & \text{if object } j \text{ is not cacheable and there are } k \text{ cacheable objects on the path from object } i \text{ to } j; \\ 0 \text{ otherwise}. \end{cases} \]

Note that \( \sum_{k=0}^{d_{ij}} W_{ijk} = 1 \).

Table 2: Helper Variables for Linear Formulation

by \( \sum_{j \in \text{subtree}(i)} \sum_{k=1}^{d_{ij}} Z_{ijk} (1 - h)^k I_{gi} \). Finally, the intrinsic generation cost of all non-cacheable objects in the subtree rooted at \( i \) can be expressed as \( \sum_{j \in \text{subtree}(i)} \sum_{k=0}^{d_{ij}} W_{ijk} (1 - h)^k I_{gi} \).

Putting this all together gives us the following expression for the total generation cost for object \( i \in J \):

\[
C_{gi} = \sum_{j \in \text{subtree}(i)} \sum_{k=1}^{d_{ij}} Z_{ijk} (1 - h)^{(k-1)} h C_{fi} + \sum_{j \in \text{subtree}(i)} \sum_{k=1}^{d_{ij}} Z_{ijk} (1 - h)^k I_{gi} \\
+ \sum_{j \in \text{subtree}(i)} \sum_{k=0}^{d_{ij}} W_{ijk} (1 - h)^k I_{gi}
\]

### 3.3.2 Benefit of Marking an Object

Certain objects are “better” candidates for caching than others. We wish to capture the value or benefit \( b_i \) that can be obtained by marking an object \( i \) cacheable. The key factors that determine \( b_i \) are the lifetime, access rate, and number of object instances, as discussed below.

**Lifetime:** Let \( t_i \) denote the lifetime of object \( i \). If the lifetime is too small, the potential reuse of the object will be very low and there is little benefit in marking the object as cacheable. Intuitively, the benefit varies logarithmically with the lifetime of an object. For example, if an object’s lifetime is increased from 1 second to 1 hour, there will be a large improvement in benefit. However, if the lifetime is increased from 1 day to 2 days, the benefit improvement is marginal.

**Access rate:** Let \( F_i \) denote the access rate of an object \( i \). \( F_i \) indicates how many times some instance of object \( i \) is accessed per unit time. The benefit of marking an object increases with the access rate \( F_i \). For example, if there are two objects \( A \) and \( B \) having access rates of 2 per second and 10 per second, respectively, then caching \( B \) will result in a higher benefit than caching \( A \).
**Number of object instances:** Let \( N_i \) be the number of instances of an object \( i \). As \( N_i \) increases, the probability that the same object instance will be accessed more than once decreases, and thus the benefit of caching also decreases. To see this, consider two objects \( A \) and \( B \) having equal access rates, i.e., \( F_A = F_B \). Suppose further that the number of instances of \( A \) is 1000 \((N_A = 1000)\), whereas the number of instances of \( B \) is 1 \((N_B = 1)\). Each time a request for \( A \) is made, there is some probability that the request is for a particular instance \( j \) of \( A \), which we denote as \( A_j \). If the distribution of requests is uniformly distributed, then each time object \( A \) is accessed, the probability that instance \( A_j \) is requested is 0.001. On the other hand, when a request for \( B \) is made, the probability that some instance \( B_k \) is requested is 1, since there is only a single instance of \( B \). In comparing these two objects for cacheability, \( B \) is a better candidate for caching since it has a greater probability of a cache hit. Object \( B \) is an example of a *singleton object*, an object having a single instance. In general, singleton objects are better candidates for caching than non-singleton objects, since they typically have higher cache hit rates.

As discussed above, the benefit of marking an object varies logarithmically with the lifetime, while the benefit is increasing in the access rate and decreasing in the number of object instances. Thus, we define the benefit of marking an object as \( b_i = \log(t_i) \cdot F_i/N_i \).

### 3.4 Constraints

There are three sets of constraints that the system must satisfy: *locality*, *performance*, and *structural* constraints. We describe each in turn.

**Locality Constraint:** An ideal system has infinite main memory and thus, all object instances can be cached in memory. In reality, however, systems have limited memory. To make the best use of this limited memory, we have to choose objects to mark such that frequently accessed object instances can be fetched from memory. The percentage of accesses that reference a particular object instance is called the *locality* of the instance. Typically, access patterns on instances follow the \( l - m \) power law. This means that \( m\% \) of the instances are accessed in \( l\% \) of the cases, where \( l\% \) represents the locality of the object. We denote the locality of an object \( i \) by \( l_i \). Ideally, we would like to store all such “hot” object instances (i.e., instances requested
frequently) in the cache.

Let $C_i$ be the storage requirement for object $i$, which depends on the object size ($S_i$), number of object instances ($N_i$), and object locality ($l_i$). Thus, $C_i$ is given by $C_i = S_iN_i l_i$.

Let $h$ be the target hit rate; i.e., $h\%$ of all requested instances should be served from cache. To achieve this hit rate requires that $h\%$ of the frequently accessed instances be stored in cache. If the total available cache size is $M$ bytes, then the total size of the object instances in the cache at any given time should not exceed $M$. This locality constraint can be expressed as follows:

$$h \sum_{i \in I} C_i X_i \leq M.$$

**Performance Constraint:** Mission critical applications typically have strict task deadlines. For instance, an online retail application may have the requirement that a catalog page request must be served to a customer within 3 seconds or less. Thus, each of the sub-tasks required to generate the page, e.g., creation and destruction of object instances, must be completed within a small enough time frame such that this requirement is met. Such performance requirements must be considered when choosing which objects to mark as cacheable.

Consider, for example, the case where we have to decide which of the two objects, $A$ and $B$, to cache. Suppose that, on average, an instance of $A$ requires 20 ms to fetch from the cache and 50 ms to generate, while an instance of $B$ requires 20 ms to fetch and 100 ms to generate. Suppose further that the application has a requirement that both object instances be created within 70 ms or less. If we choose to cache $A$ rather than $B$, then the total generation time would be 120 ms, which is unacceptable, given the 70 ms generation time requirement. On the other hand, if we choose to cache $B$ rather than $A$, the total generation time is 70 ms, which meets the generation time requirement. To capture this type of performance requirement, we associate a maximum response time $T_j$, $\forall j \in J$, with each root level object.

**Structural Constraints:** To model this problem as an integer linear program, we require certain structural constraints, as described below:

1. **An object that is cacheable must be marked as cacheable.** More formally, $Z_{ijk} = 1$ if and only if $X_j = 1$ and $Y_{ijk} = 0$. This constraint can be expressed as $Z_{ijk} \geq X_j - Y_{ijk}, \forall j \in subtree(i), \forall i \in J$ and $k = 1, \ldots, d_{ij}$.
2. An object that is not cacheable must not be marked as cacheable. More formally, $W_{ijk} = 1$ if and only if $X_j = 0$ and $Y_{ijk} = 0$. This constraint can be expressed as $W_{ijk} \geq (1 - X_j - Y_{ijk})$, $\forall j \in \text{subtree}(i)$, $\forall i \in J$ and $k = 0,...,d_{ij}$.

3. Exactly one of the $Y_{ijk}$ should be 0 for each pair of $i$ and $j$. Note that if none of the objects in between $i$ and $j$ is cacheable, then $Y_{ij0} = 0$. Thus, for each $Y_{ijk}$, we have $\sum_{k=0}^{d_{ij}} Y_{ijk} = d_{ij} - 1$, $\forall i \in J$.

### 3.5 Problem Formulation

We now present the complete formulation of our integer linear programming model. Recall that the objective of the model is to identify the objects which can be marked as cacheable so that the total cost of accessing all root objects ($\forall i \in J$) from the application layer is minimized. Thus, for each root object, we consider the generation cost of the root object and the benefit of marking that object. Since different root objects have different access rates, we normalize the cost on the basis of access rate. As a result, the objective function in our model is given by $\sum_{\forall i \in J} \{C_{gi} - X_i h b_i\} F_i$.

The integer linear program is shown in Table 3:

Minimize $\sum_{\forall i \in J} \{C_{gi} - X_i h b_i\} F_i$  
\hspace{1cm} (2)

\text{s.t.}  
\hspace{1cm} $h \sum_{\forall i \in J} C_i X_i$ \hspace{1cm} $\leq M$  
\hspace{1cm} $\sum_{\forall i \in J} \{C_{gi} - X_i h b_i\}$ \hspace{1cm} $\leq T_i$ \hspace{1cm} $\forall i \in J$  
\hspace{1cm} $X_j - Y_{ijk}$ \hspace{1cm} $\leq Z_{ijk}$ \hspace{1cm} $\forall j \in \text{subtree}(i)$, $k = 1,...,d_{ij}, \forall i \in J$  
\hspace{1cm} $(1 - X_j - Y_{ijk})$ \hspace{1cm} $\leq W_{ijk}$ \hspace{1cm} $\forall j \in \text{subtree}(i)$, $k = 0,...,d_{ij}, \forall i \in J$  
\hspace{1cm} $\sum_{k=0}^{d_{ij}} Y_{ijk} + 1$ \hspace{1cm} $d_{ij}$ \hspace{1cm} $\forall j \in \text{subtree}(i), \forall i \in J$  
\hspace{1cm} $\sum_{j \in \text{subtree}(i)} \sum_{k=1}^{d_{ij}} Z_{ijk} (1 - h)^{(k-1)} h C_{fi}$ \hspace{1cm} $= C_{gi}$ \hspace{1cm} $\forall i \in I$  
\hspace{1cm} $\sum_{j \in \text{subtree}(i)} \sum_{k=1}^{d_{ij}} Z_{ijk} (1 - h)^k I_{gi}$ \hspace{1cm} $= 0$ \hspace{1cm} $\forall i \in I$  
\hspace{1cm} $\sum_{j \in \text{subtree}(i)} \sum_{k=0}^{d_{ij}} W_{ijk} (1 - h)^k I_{gi}$ \hspace{1cm} $= 0$ or 1 \hspace{1cm} $\forall i \in I$  

Table 3: IP Formulation

Constraint set (3) ensures that the available memory is not exceeded, and constraint set (4) ensures that the response time for object $i$ is less than the maximum acceptable response
time. Constraint sets (5-7) are the structural constraints, while constraint sets (8-9) compute the generation costs for the objects.

4 Solution Approach

To find an exact solution to the problem presented in Section 3, one can use an appropriate solver for integer programming, such as CPLEX [14]. However, due to its size and complexity, finding an exact solution may not always be a practical alternative. In a typical enterprise application, the number of objects is on the order of thousands, and thus, finding the optimal solution would require at least several hours and very often, longer. Moreover, the decision makers who would need to solve this problem, e.g., system designers, would very likely not have access to solver tools such as CPLEX. Given the difficulties associated with exact solution methods, in this section, we discuss a heuristic approach. To test the performance of the heuristic, we compare its results to a lower bound obtained from the linear programming (LP) relaxation of the IP. Before doing so, we first show that the problem is NP-complete.

4.1 Proving NP-Completeness

We prove the NP-completeness of our cacheability problem by first showing that it belongs in NP, and then reducing the knapsack problem [11] to a special case of our problem.

The cacheability problem is easily seen to be in NP. In this problem, for each object \( i \), we are assigning the cacheability variable, \( X_i \), to be either 0 or 1. If the total number of objects (i.e., the cardinality of set \( I \)) is \( N \), then the number of possible outcomes for this problem is \( O(2^N) \), which is exponential with respect to \( N \). Moreover, none of the non-structural constraints, i.e., constraint sets (3) and (4), alone can reduce the solution space of the problem. Thus, the solution space of this problem grows exponentially with the number of objects. Therefore, our cacheability problem is in NP.

To show that the knapsack problem reduces to a special case of our cacheability problem, we consider the following instance of our problem. The object hierarchy is a single-level hierarchy having one root object, object 0, and \( n \) objects at the first level of the hierarchy. These \( n \)
objects are immediate children of the root object and are independent of one another, i.e., there are no edges representing contained by relationships among the \( n \) objects. For the root object, object 0, the intrinsic generation cost, size, and benefit of marking the object are all zero; i.e., \( I_{g_0} = 0, S_0 = 0, \) and \( b_0 = 0, \) respectively. For all objects, the fetch cost is zero, i.e., \( C_{f_i} = 0, \forall i \in I. \) The maximum acceptable response time for generating a deliverable object \( i \) is infinite, i.e., \( T_i = \infty, \forall i \in J. \) Given this instance of our problem, the generation cost evaluates to

\[
C_{g_i} = X_i I_{g_i} (2 - h) - I_{g_i}.
\]

The objective function then becomes

\[
\sum_{i=1}^{n}(X_i I_{g_i} (2 - h) - X_i h b_i - I_{g_i}) F_i.
\]

Letting \( K = (h b_i - I_{g_i} (2 - h)) F_i \), the problem can be expressed as follows (note that constraint set (4) can be dropped since \( T_i \) is infinite):

\[
\text{Maximize } \sum_{i=1}^{n} K_i X_i \quad \text{s.t.:} \quad \sum_{i=1}^{n} C_i X_i \leq M/h \quad (11)
\]

This problem is equivalent to the knapsack problem. Given that our cacheability problem belongs in NP and the knapsack problem reduces to a special case of our problem, we know that our cacheability problem is NP-complete.

### 4.2 LP Relaxation

A well-known technique used to find bounds on the optimal solution for IPs is to use linear programming (LP) relaxation. With this technique, an LP is generated from an IP by replacing the integer variables with appropriate continuous variables. Applying this technique to our formulation in Section 3 would involve replacing constraint set (10) by the following constraint set: \( 0 \leq X_i \leq 1, \forall i \in I. \) The solution to the LP relaxation provides a lower bound on the optimal integer solution (of the minimization problem). This lower bound solution will be used to compare the accuracy of our heuristic approach in Section 5.

### 4.3 Heuristic Solution

We now present a heuristic algorithm that can be used to solve the cacheability problem. As our results in Section 5 will show, the heuristic produces solutions that are reasonably close to optimal. Our heuristic is based on two important criteria associated with objects: reusability, and fetch to generation ratio. We describe each in turn.
Reusability. The reusability of an object increases as the object’s lifetime, access rate, and locality increase, while the reusability decreases as the number of object instances increases (refer to Section 3). Based on these factors, we define reusability $R_i$ of an object $i$ as $R_i = \frac{t_i F_i}{N_i t_i}$. The higher the $R$ value, the greater the cacheability of an object.

Fetch to Generation Ratio (FTGR). We define the Fetch to Generation Ratio (FTGR) as the ratio of the generation cost of an object that is marked cacheable to the generation cost of an object that is not marked cacheable. Thus, FTGR is given by $FTGR = \frac{h^*C_i + (1-h^*)\gamma_i}{\gamma_i}$. The lower the $FTGR$ value, the higher the cacheability of an object.

As shown in Algorithm 1, the heuristic consists of four procedures: Initialize, Prune, Rank, and Exchange, described next.

The Initialize procedure (shown in Algorithm 2) computes $FTGR$ for each object (Steps 8-10 of Initialize). The Prune procedure (shown in Algorithm 3) applies the following simple rules to decide the cacheability of objects:

1. Objects that do not take much space in memory relative to other objects and that have lower fetch costs than intrinsic generation costs are marked cacheable (Steps 1-5 of Prune). The objective of this rule is to improve the response time of the system.

2. If after marking some of the objects cacheable (by applying the first pruning rule), we find that a root object is meeting the response time goal, then we do not need to consider the objects which are contained by this root object (Steps 6-12 of Prune).

3. Mark the remaining objects cacheable if the sum of the sizes of all the objects is less than the total memory available (Steps 13-20 of Prune).

Algorithm 1 Heuristic Algorithm to Determine Cacheability

1: Initialize()
2: Prune()
3: Rank()
4: Exchange()

If, after the Prune procedure, there is still space available in memory, the algorithm proceeds to the Rank procedure (shown in Algorithm 4), where the cacheability of the remaining objects is determined. The algorithm first computes the reusability of the remaining objects (Steps 1-4 of Rank). Next, these objects are ranked in descending order of reusability $R$ (Step 7 of
\textbf{Algorithm 2} Initialize Procedure

1: /* Initialize flag to indicate whether an object’s cacheability has been determined */
2: /* $P_i = 1$ indicates the object’s cacheability has been determined, $P_i = 0$ otherwise */
3: for $\forall i \in I$ do
4: \hspace{1em} $P_i = 0$
5: for $\forall i \in I$ do
6: /* Initialize $X_i$ values */
7: \hspace{1em} $X_i = 0$
8: for $\forall i \in I$ do
9: /* Calculate $FTGR$ for all objects */
10: \hspace{1em} $FTGR_i = \frac{h * C_{fi} + (1 - h) * I_{gi}}{I_{gi}}$

\textbf{Algorithm 3} Prune Procedure

1: /* Cache all objects which do not require much memory to store and which have lower fetch cost than generation cost */
2: for $\forall i \in I$ do
3: \hspace{1em} if $\frac{\sum_{j \in \text{subtree}(j)} (I_{gi} * X_i + (1 - X_i) * (h * C_{fi} + (1 - h) * I_{gi}))}{1 + \sum_{k \neq j} I_{gi}} < t_j$ & $FTGR_i < 1$ then
4: \hspace{2em} $X_i = 1$
5: \hspace{1em} $P_i = 1$
6: /* Check all root objects to determine whether response time requirement is met. Objects under such root objects are marked as non-cacheable. */
7: for $\forall j \in J$ do
8: \hspace{1em} if $\sum_{i \in \text{subtree}(j)} (I_{gi} * X_i + (1 - X_i) * (h * C_{fi} + (1 - h) * I_{gi})) < t_j$ then
9: \hspace{2em} for $\forall i \in I$ do
10: \hspace{3em} if $i \in \text{subtree}(j)$ & $!(i \in \text{subtree}(k) : k \neq j)$ then
11: \hspace{4em} $X_i = 0$
12: \hspace{3em} $P_i = 1$
13: /* Calculate residual memory $\mathcal{R}_{SM}$ */
14: $\mathcal{R}_{SM} = M - \sum_{i \in I} S_i * N_i * l_i * h * X_i$
15: /* If remaining objects can fit into available memory, then mark remaining objects as cacheable */
16: if $\sum_{i \in I} S_i * N_i * l_i * h * P_i < \mathcal{R}_{SM}$ then
17: \hspace{1em} for $\forall i \in I$ do
18: \hspace{2em} if $P_i = 1$ then
19: \hspace{3em} $X_i = 1$
20: return $X$

Rank) and in ascending order of $FTGR$ (Step 8 of \textbf{Rank}). Then, using these two rank values, namely $\mathcal{R}_{Rank}$ and $\mathcal{R}_{RankFTGR}$, the total rank for cacheability of an object is computed as $\mathcal{R}_{cacheability} = \mathcal{R}_{Rank} * \mathcal{R}_{RankFTGR}$ (Step 9 of \textbf{Rank}). Finally, the cacheability decision is made (Step 11 of \textbf{Rank}) by choosing the top few objects according to the total cacheability rank, as long as the memory constraints described in Section 3 are satisfied.

After assigning the initial values of the $X_i$s based on cacheability rank (in the \textbf{Rank} procedure), the algorithm proceeds to the Exchange procedure (shown in Algorithm 5), where an exchange-based heuristic is applied. First, both the cacheable objects and non-cacheable objects (i.e., objects for which $X_i = 0$) are placed in descending order based on cacheability rank (Step 1 of
Algorithm 4 Rank Procedure

1: /* Compute reusability for each object */
2: for \( \forall i \in I \) do
3: if \( P_i = 0 \) then
4: \( R_i = \frac{F_i}{\sum_{j} F_j} \)/* Compute cacheability rank for remaining objects */
5: for \( \forall i \in I : P_i = 1 \) do
6: Compute \( \text{Rank}_R \) on the basis of descending order of \( R_i \) values
7: Compute \( \text{Rank}_{FTGR} \) on the basis of ascending order of \( FTGR_i \) values
8: \( \text{Rank}_{cacheability} = \text{Rank}_R \times \text{Rank}_{FTGR} \)
9: /* Decide cacheability based on cacheability rank */
10: Make \( X_i = 1 \) for first \( m \) objects in \( \text{Rank}_{cacheability} \) order for which memory constraint is not violated

Exchange). Next, we compute the value of the objective function (2) using the current values of the \( X_i \)'s (Step 2 of Exchange). We exchange one of the lower 10% cacheable objects with one of the upper 10% non-cacheable objects and then check to see whether this exchange violates any of the constraints (Steps 6-8 of Exchange). If no constraints are violated, we compute the new objective function value and compare this value with the original objective function value. If the new objective value is less than the original objective value, we keep this exchange (Steps 9-14 of Exchange); otherwise, we reject this exchange (Steps 16-17 of Exchange).

Algorithm 5 Exchange Procedure

1: Let \( K \) be the ordered set of the \( m \) objects ordered by \( \text{Rank}_{cacheability} \) and let \( L \) be the ordered set of all other objects (i.e., objects for which \( X_i = 0 \)).
2: Compute value of objective function (2) for current values of \( X_i \)'s; let this value be denoted by \( OBJ_{init} \).
3: /* apply exchange-based heuristic */
4: for \( i = 0 \) to \( q \) do
5: /* for some predetermined number \( q \), exchange one object of \( K \) with that of \( L \) */
6: Randomly select an object \( k \in K \) from lower 10% of set \( K \) when objects in \( K \) are ordered in ascending order of \( \text{Rank}_{cacheability} \)
7: Randomly select an object \( l \in L \) from upper 10% of set \( L \) when objects in \( L \) are ordered in ascending order of \( \text{Rank}_{cacheability} \)
8: Make \( X_k = 0 \) and \( X_l = 1 \)
9: if Memory and Response Time constraints are not violated then
10: Compute value of objective function (2) for current values of \( X_i \)'s. Let this value be denoted by \( OBJ_{exch} \).
11: if \( OBJ_{exch} < OBJ_{init} \) then
12: Put object \( l \) in \( K \) and \( k \) in \( L \).
13: Assign \( X_k = 0 \) and \( X_l = 1 \) permanently
14: \( OBJ_{init} = OBJ_{exch} \)
15: else
16: \( X_k = 1 \)
17: \( X_l = 0 \)
18: Return \( X \)

The exchange process is repeated until we obtain no more improvement in the objective function value for a few consecutive iterations, or until a maximum number of exchanges (input
to the procedure) is completed.

5 Performance Results

In this section, we examine the effectiveness of our cacheability model. First, using a real world application, we compare the cacheability decisions made by (i) expert system developers, (ii) an exact solution approach, and (iii) our heuristic solution approach. Next, using synthetic data, we examine the accuracy and running time of our heuristic.

5.1 Comparison of Cacheability Decisions using Real World Application

For this set of experiments, we chose the Web site of a major office supply retailer, as described in Section 2.2. To keep the scope of the experiments reasonable (e.g., in terms of number of objects), we used only the home page of the site (refer to Figure 2[B]), and considered a 10 MB cache storage. Note that 10 MB of cache storage may seem to be very small with respect to large enterprise systems, which typically have memory sizes in the GB range. However, this is an appropriate size for our experiments since we have concentrated our study on a very small part of a large Web site. Thus, we have scaled down the available resources to simulate a realistic caching system - i.e., a system having constrained memory for cache storage. The relevant parameters associated with these objects are provided in Table 6 in the Appendix.

We engaged a team of expert system developers to decide the cacheability of these objects on the basis of the data in Table 6. This team was responsible for the design and development of the retailer’s Web application, and hence had significant knowledge of the objects and their properties. The expert team applied intuition based on their knowledge and experience and decided on the cacheability solution shown in Table 4 in the column labeled “Expert”. In Table 4, “C” denotes a decision to cache, whereas “NC” denotes a decision not to cache.

The “Expert” caching scheme achieved a 45% cache hit ratio. Next, we solved the model presented in Section 3 with an object response time of 2 seconds for the home page and a cache hit rate of 80%. The solution (found by CPLEX) is shown in Table 4 in the column labeled
“Exact”. Finally, we ran our heuristic using the same parameters. The heuristic yielded the same results as the exact solution method (refer to the column labeled “Heuristic” in Table 4). Thus, all three approaches resulted in similar solutions. However, the expert solution included two additional objects, the Hot-Deal and Great-Deal objects.

<table>
<thead>
<tr>
<th>Object</th>
<th>Expert</th>
<th>Exact</th>
<th>Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homepage</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Top Navigation Bar (TNB)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Right Navigation Bar (RNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Left Navigation Bar (LNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Middle Content (MC)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Bottom Navigation Bar (BNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Personalized Cart (LNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Navigation Bar (TNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Advertisement (TNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Advertisement (RNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Special Offer (RNB)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Articles (RNB)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>General Info (LNB)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Category (LNB)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Sub-category (LNB)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Products (MC)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Free-Offer (MC)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Hot-Deal (MC)</td>
<td>C</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Great-Deal (MC)</td>
<td>C</td>
<td>NC</td>
<td>NC</td>
</tr>
<tr>
<td>Product Detail (MC)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Related Products (MC)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Navigation Bar (BNB)</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Personalized Content (BNB)</td>
<td>NC</td>
<td>NC</td>
<td>NC</td>
</tr>
</tbody>
</table>

Table 4: Comparison of Cacheability Decisions

We performed a set of experiments to compare the cacheability decisions in Table 4 based on run-time system performance of the office supply retail Web site. User traffic on the site was simulated using the virtual user generator of a standard load generating tool (Mercury Interactive’s LoadRunner [15]). Then, as the load on the site was varied, the average response time was recorded. Response time measures the total round trip time of a user’s request. Object caching reduces the page generation time, which in turn, reduces the end-to-end response time that an end user experiences. Thus, average response time is an appropriate metric for these experiments.

The office supply site home page takes as input the following parameters (passed through the URL): (i) Category List - Sub-categories of the categories that are displayed on the requested Web page, (ii) Sub-Category List - Products of the sub-categories that are displayed, (iii) Random
Advertisement Number - Random number used to select an advertisement (since advertisements are randomly selected, this object has been assigned a locality of 0, as shown in Table 6), and (iv) User's Interest List - Denotes the profile of a user.

To obtain the request parameters for our experiments, we analyzed the Web site log for one day and accumulated the URLs for the site home page. These URLs, which contain the request parameters, were fed into the LoadRunner tool. LoadRunner's virtual user generator sequentially picks up these URLs and sends a request to the site. The response time data was accumulated by LoadRunner's Controller module.

Figure 4[A] shows the response time (in seconds) for three cases as load (in clicks per second) is varied from 10 to 100 clicks per second: (i) no cache, where no caching is employed, (ii) optimal, where the model is solved for an exact solution using CPLEX, and (iii) expert, where the decision is made using the intuition of system experts. As shown in Figure 4[A], all three curves are increasing with load. However, the improved performance due to caching is clear - the curves corresponding to the optimal and expert cases are much lower and increase much more slowly than the no cache case. In particular, the no cache case has a response time of 5 seconds at a load of 10 clicks per second, while for the expert case, the response time is only 2.2 seconds. The optimal case provides further improvement, with a response time of 1.2 seconds. This difference in performance increases as the system load increases. For instance, at a load of 60 clicks per second, the response time of the no cache case is 17 seconds, 4.5 seconds for the expert case and 1.7 seconds for the optimal case.

The expert caching scheme resulted in only a 45% cache hit ratio, whereas the optimal caching scheme resulted in a 74% cache hit ratio. Recall that the objective cache hit ratio used in the model was 80%. The reduction of the hit ratio value from 80% to 74% is due to the mathematical approximation required to gather data and to build the model.

5.2 Analysis of Heuristic Solution Method using Synthetic Data

Using synthetic data, we analyze the heuristic presented in Section 4 based on two criteria: (i) running time, the time required to find a solution, and (ii) accuracy, a measure of the deviation
of the solution obtained from a lower bound on the optimal solution. Unless stated otherwise, parameter values used in this section are the same as those used in the previous section.

### 5.2.1 Running Time

Before we compare the running times for the exact and heuristic solution methods numerically, we analyze the running time complexity of our heuristic.

The for loops in lines 3, 5, and 8 of the Initialize procedure, and the for loop in line 2 of the Prune procedure are each of order $O(N)$, where $N$ is the total number of objects (i.e., the cardinality of the set $I$). The for loop in line 7 of the Prune procedure is of order $O(N_r)$, where $N_r$ is the total number of deliverable objects (i.e., the cardinality of the set $J$). The for loop in line 2 of the Rank procedure is $O(N)$. Ordering of the objects according to $R_i$s (line 7) and ordering the objects according to $FTR_i$s (line 8 of the Rank procedure) are $O(log(N))$. The for loop in line 6 of the Rank procedure is $O(N)$. The cacheability assignments made on line 11 of Rank require $N$ operations in the worst case and so this step is $O(N)$. The for loop in line 4 of the Exchange procedure is $O(q)$, where $q$ is the total number of iterations through the Exchange phase. Therefore, the running time complexity of the heuristic algorithm is $O(N + log(N) + q)$. Since $N >> q$, the total running time complexity of the heuristic algorithm is $O(N)$.

Figure 4[B] shows a comparison of the running times (in seconds) for the exact (labeled optimal) and heuristic solution methods as the number of objects is varied from 5 to 20. Running
times are shown using a log scale. While the running times for both solution methods are increasing in the number of objects, the heuristic method requires orders of magnitude less time to solve than the exact method.

5.2.2 Accuracy

To measure the accuracy of our heuristic algorithm, we compute the percentage difference in the solution obtained using the heuristic method with the lower bound solution (obtained using the LP relaxation in Section 4.2). Figure 5 shows this percentage difference as the number of objects is varied from 50 to 2000.

As Figure 5 shows, the heuristic method provides reasonably accurate solutions, with the gap varying from about 4.7% to about 6.1% over the range studied. These results imply that the heuristic approach will produce a solution that is within 6% of the lower bound solution in most cases. Thus, given its good running time performance and accuracy, the heuristic method appears to be a solid choice for solving the cacheability problem for larger problem sizes.

6 Related Work

As far as we are aware, there is no other published work that addresses the design-time cacheability decision problem. Thus, we briefly review the work that is most closely related to ours, which includes fragment-level caching, cache replacement, and OR/MS techniques applied to caching problems.
Fragment-level caching refers to caching objects, such as presentation or application-layer objects (e.g., [30]). Commercial fragment-level caching solutions include EdgeSuite from Akamai [28], Oracle 9i Web Cache [6], as well as several application server products, such as WebLogic from BEA Systems [26] and WebSphere from IBM [13].

As mentioned previously, the run-time or cache replacement problem has been studied extensively in the operating system literature (e.g., [19, 31, 4]). This problem has also been studied in the context of web proxy caching (e.g., [29, 1, 31]).

Though our design-time cacheability problem has not been addressed in the OR/MS literature, there are several other problems related to caching that the OR/MS community has studied. For instance, the optimal location of caches in a network has been modeled using dynamic programming approaches [18, 5]. Another related problem is the object replication problem, which has been modeled as a combinatorial optimization problem [17]. The cache replacement problem has been modeled as a knapsack problem in [2].

It is worth mentioning that we have addressed this problem in a recent work [9]. In particular, we model the problem as a non-linear program and propose a solution technique based on genetic algorithms (GA). While [9] provides an approach to solving this problem, it has many limitations. For example, [9] only considers a non-linear formulation (suffering from the drawbacks discussed previously), whereas the present work proposes a linear formulation. In addition, [9] proposes a GA-based solution approach, which is difficult to deploy. In the present work, we propose a heuristic solution approach, which is simple to deploy. Yet another limitation of [9] is that it does not address the performance improvements made possible by applying the proposed technique. In the present work, we provide a set of performance results, using both synthetic and real-world data, which illustrate the performance improvements achievable. Finally, in the present work, we show that the problem is NP-complete, justifying the use of a heuristic approach.

7 Conclusion

OO technologies have become widely adopted in enterprise applications due to the additional functionality and flexibility they provide to these applications. At the same time, however,
OO technologies also require significant amounts of computational power to support, greatly impacting the performance and scalability of such applications. Object caching is a common solution to this problem. We have shown how the application of object caching maps into an optimization problem. In particular, we have focused on the design-time decision of determining which objects should be candidates for caching. To the best of our knowledge, ours is the first work to address this issue. We have formulated this problem as a linear integer program (IP) and presented a heuristic solution approach. We have shown that our heuristic is efficient, having a running time complexity that is linear in the number of objects. We have also demonstrated, through a set of experiments, that our heuristic provides solutions that are reasonably close to optimal. We believe that our proposed solution can be an integrated part of OO design products, such as Rational Rose [24] and TogetherJ [25].

References


Appendix

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I$</td>
<td>Set of objects such that object $i \in I$</td>
</tr>
<tr>
<td>$J$</td>
<td>Set of deliverable objects such that $j \in J$, $J \subseteq I$</td>
</tr>
<tr>
<td>$D(i)$</td>
<td>Set of objects that are contained by $i$, $d(i) \in D(I)$</td>
</tr>
<tr>
<td>$F_j$</td>
<td>Frequency of access of object $j$, $\forall j \in J$</td>
</tr>
<tr>
<td>$S_i$</td>
<td>Size of object $i$</td>
</tr>
<tr>
<td>$l_{si}$</td>
<td>Intrinsic size of object $i$</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Number of instances of object $i$</td>
</tr>
<tr>
<td>$l_i$</td>
<td>Locality of object $i$</td>
</tr>
<tr>
<td>$t_i$</td>
<td>Lifetime of object $i$</td>
</tr>
<tr>
<td>$C_i$</td>
<td>Storage required for object $i$, $C_i = S_i \cdot N_i \cdot l_i$</td>
</tr>
<tr>
<td>$h$</td>
<td>Hit ratio for the cache ($0 \leq h \leq 1$)</td>
</tr>
<tr>
<td>$b_i$</td>
<td>Benefit of marking object $i$, $b_i = \log(t_i) \cdot F_i/N_i$</td>
</tr>
<tr>
<td>$M$</td>
<td>Size of cache</td>
</tr>
<tr>
<td>$C_{gi}$</td>
<td>Cost of generating an instance of object $i$</td>
</tr>
<tr>
<td>$l_{gi}$</td>
<td>Intrinsic generation cost of an instance of object $i$</td>
</tr>
<tr>
<td>$C_{fi}$</td>
<td>Cost of fetching object $i$ from the cache</td>
</tr>
<tr>
<td>$T_j$</td>
<td>Objective response time for deliverable object $j$, $j \in J$</td>
</tr>
<tr>
<td>$d_j$</td>
<td>Maximum depth of the tree rooted at object $j \in J$</td>
</tr>
<tr>
<td>$\text{subtree}(j)$</td>
<td>Set of all nodes in $I$ which are directly or indirectly contained by object $j \in J$</td>
</tr>
</tbody>
</table>

Table 5: Notation
<table>
<thead>
<tr>
<th>Object</th>
<th>Intrinsic Size (KB)</th>
<th>Number of Instances</th>
<th>High Locality</th>
<th>Lifetime (sec)</th>
<th>Intrinsic Generation Cost (ms)</th>
<th>Fetch Cost (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HomePage</td>
<td>1</td>
<td>1,000,000</td>
<td>82</td>
<td>0</td>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>Top Navigation Bar (TNB)</td>
<td>1,000,000</td>
<td>70</td>
<td>0</td>
<td>150</td>
<td>80</td>
<td>110</td>
</tr>
<tr>
<td>Right Navigation Bar (RNB)</td>
<td>10,000</td>
<td>0</td>
<td>5</td>
<td>80</td>
<td>90</td>
<td>115</td>
</tr>
<tr>
<td>Left Navigation Bar (LNB)</td>
<td>10</td>
<td>100,000</td>
<td>85</td>
<td>3600</td>
<td>150</td>
<td>120</td>
</tr>
<tr>
<td>Middle Content (MC)</td>
<td>2</td>
<td>1,000,000</td>
<td>50</td>
<td>3600</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Bottom Navigation Bar (BNB)</td>
<td>0.8</td>
<td>1,000,000</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>Personalized Cart (TNB)</td>
<td>0.1</td>
<td>1,000,000</td>
<td>0</td>
<td>0</td>
<td>105</td>
<td>20</td>
</tr>
<tr>
<td>Navigation Bar (TNB)</td>
<td>1.5</td>
<td>10</td>
<td>90</td>
<td>60</td>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td>Advertisement (TNB)</td>
<td>1.8</td>
<td>1000</td>
<td>0</td>
<td>5</td>
<td>150</td>
<td>60</td>
</tr>
<tr>
<td>Advertisement (RNB)</td>
<td>2.0</td>
<td>100</td>
<td>0</td>
<td>5</td>
<td>150</td>
<td>60</td>
</tr>
<tr>
<td>Special Offer (RNB)</td>
<td>4.0</td>
<td>10</td>
<td>80</td>
<td>3600</td>
<td>175</td>
<td>50</td>
</tr>
<tr>
<td>Articles (RNB)</td>
<td>5.5</td>
<td>10</td>
<td>80</td>
<td>3600</td>
<td>120</td>
<td>40</td>
</tr>
<tr>
<td>General Info (LNB)</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>3600</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>Category (LNB)</td>
<td>8</td>
<td>100</td>
<td>85</td>
<td>3600</td>
<td>650</td>
<td>100</td>
</tr>
<tr>
<td>Sub-category (LNB)</td>
<td>10</td>
<td>1000</td>
<td>77</td>
<td>3600</td>
<td>750</td>
<td>120</td>
</tr>
<tr>
<td>Products (MC)</td>
<td>10</td>
<td>1,000,000</td>
<td>50</td>
<td>3600</td>
<td>90</td>
<td>110</td>
</tr>
<tr>
<td>Free-Offer (MC)</td>
<td>15</td>
<td>100</td>
<td>80</td>
<td>3600</td>
<td>250</td>
<td>150</td>
</tr>
<tr>
<td>Hot-Deal (MC)</td>
<td>5</td>
<td>1000</td>
<td>80</td>
<td>3600</td>
<td>60</td>
<td>190</td>
</tr>
<tr>
<td>Great-Deal (MC)</td>
<td>5</td>
<td>1000</td>
<td>80</td>
<td>3600</td>
<td>60</td>
<td>210</td>
</tr>
<tr>
<td>Product Detail (MC)</td>
<td>10</td>
<td>100,000</td>
<td>80</td>
<td>3600</td>
<td>600</td>
<td>120</td>
</tr>
<tr>
<td>Related Products (MC)</td>
<td>10</td>
<td>10,000</td>
<td>80</td>
<td>3600</td>
<td>700</td>
<td>130</td>
</tr>
<tr>
<td>Navigation Bar (BNB)</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>3600</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>Personalized Content (BNB)</td>
<td>0.1</td>
<td>1,000,000</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 6: Parameters for Objects