Management of Speedup Mechanisms in Learning Architectures

John Cheng

January 1995
CMU-CS-95-112

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:

Tom Mitchell, Chair
Jaime Carbonell
Jill Lehman
John Laird

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory or the United States Government.

Abstract

Learning architectures typically operate rather inefficiently. To increase performance, two strategies are commonly used: speedup mechanisms are incorporated into the architecture, and architecture operation is simplified. Unfortunately, both these strategies have drawbacks.

Because of the utility problem, inappropriate use of speedup mechanisms can actually decrease system efficiency. Hence, good speedup mechanism management – deciding when, where, and which speedup mechanism to use – is important if the mechanisms are to be effective. Typically, however, good management strategies are not available. Architecture-provided strategies are usually very simple, and cannot use the mechanisms appropriately all the time. Good user-provided strategies are also difficult to develop – under a complex system or domain, it can be difficult to understand system behavior well enough to specify a good management strategy. Furthermore, user or architecture-provided management techniques are usually fixed, and cannot adapt to environment dynamics. Hence, lack of good management strategies limit the effectiveness of speedup mechanisms.

Simplifying an architecture's inference mechanism yields dramatic efficiency gains. Unfortunately, gaining efficiency in this manner usually sacrifices fine-grain control over the behavior of the system, or *architecture flexibility*. Consequently, this speedup technique forces the domain designer to operate at the flexibility/efficiency tradeoff point chosen by the architecture.

This thesis investigates ways of handling both of these problems. The speedup mechanism management problem is approached by making the architecture itself responsible for developing a management strategy. An agent, embedded into the architecture, observes system operation, invoking speedup mechanisms appropriately. This approach allows the architecture to tailor its strategies individually to different domains, increasing speedup mechanism usefulness. Furthermore, because the agent can monitor the architecture continuously, it can adapt its management strategies to the dynamics of the environment.

This dissertation also presents an algorithm that can be used to reduce the flexibility/efficiency constraints on the domain designer, giving him more options. The designer is allowed architecture flexibility, but if flexibility is not needed, the unnecessary flexibility is automatically traded for efficiency.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

As artificial intelligence has evolved, integrated architectures – architectures that unify problem solving and learning methods, such as Soar [Laird 87], Prodigy [Minton 87] and Theo [Mitchell 91] – have become very important research tools. To be successful, an architecture must be flexible and powerful enough to handle many domains with widely varying characteristics and requirements. Unfortunately, such general-purpose architectures often perform inefficiently. To increase performance, architecture designers have used two strategies:

- Integrate speedup mechanisms with architectures
- Reduce architecture flexibility or functionality

The advantages and disadvantages of each of these tactics is discussed below.

1.1.1 Speedup Mechanisms

Adding speedup mechanisms – such as caching, explanation-based generalization (EBG) [Mitchell 86], and chunking [Laird 87] – to architectures is very common, and can be a very effective way of increasing system efficiency. Virtually all architectures support one or several speedup mechanisms. Unfortunately, because of the *utility problem*, the existence of speedup mechanisms is no panacea. In its broadest form, the utility problem states that inappropriate use of speedup mechanisms can decrease system performance.

Hence, using them can backfire. To realize the potential of speedup mechanisms, it is necessary to use them *appropriately*. Hence there must be an effective *management strategy* determining when, where, and which mechanisms to apply.

How do architectures typically handle speedup mechanism management? Often, they rely on simple, naive, but generally effective schemes. For instance, the Theo architecture supports caching and EBG. In general, caching increases Theo performance. Hence, by default, Theo caches all query instances. On the other hand, EBG in general decreases Theo performance. If EBG is applied at every opportunity, the success rates of EBG rules tend to be so low that rule application costs dominate the expected inference time saved by the rules. Therefore, by default, Theo never uses EBG. The drawback to this kind of management strategy is that, obviously, speedup mechanisms are not being used appropriately all of the time. Although EBG usually slows Theo down, there are situations for which EBG can drastically increase system performance. Likewise, for some situations, caching can significantly decrease system performance.

Architectures can also rely on humans to determine a speedup mechanism management strategy. I.e., users or knowledge-base designers specify the situations under which various speedup mechanisms should be invoked. It can be almost impossible to specify a good management scheme, however. Under a complex system or domain, it may be difficult to understand system behavior well enough to specify a good management strategy. It can be especially difficult to manage multiple speedup mechanisms. If several are applicable in a situation, which one should be applied? Should they all be applied? Can the mechanisms interact in some way, and if so, what are the ramifications of this interaction? Finally, and most importantly, architecture environment factors – such as query distributions – are dynamic, often changing over time, making any fixed speedup strategies obsolete.

Because it is so difficult to manage speedup mechanisms effectively, integrating them with learning architectures does not necessarily result in increased architecture performance. The management strategies discussed above are too coarse-grained and inflexible to fully utilize the potential of speedup mechanisms.

1.1.2 Reducing Architecture Flexibility

"Architecture flexibility" is the *configurability* of the architecture. For example, assume that the user is interested in computing the *area* of a variety

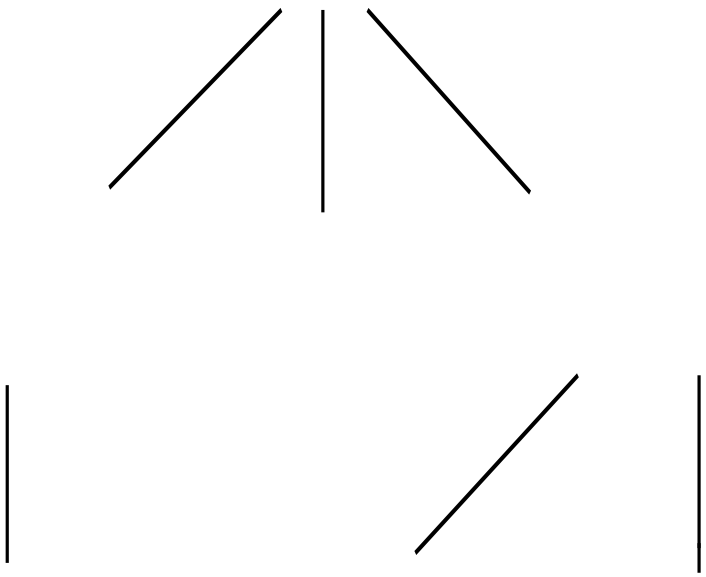

Figure 1.1: An Object/Class Hierarchy

of different objects. For some sets of objects, the user might find it useful to be able to specify different formulas for different classes of objects. A *flexible* architecture allows this. Consider the object/class hierarchy in Figure 1.1. Theo, which is a very flexible architecture, allows the user to specify different inference methods (i.e., formulas) for every object class – for every node in the hierarchy – so the user can use different formulas for circles, triangles, and squares. On the other hand, an inflexible architecture such as TheoGT [Mitchell 93], allows the user to specify only *one* inference method for the entire object/class hierarchy. This can be very inconvenient for the user – it is difficult for the user to specify the behavior of the architecture. This example shows *inference method* flexibility – only one of many dimensions of flexibility.

Reducing architecture flexibility increases architecture performance. Why is this so? Consider inference method flexibility. Since a flexible architec-

ture supports multiple inference methods for every object/class hierarchy, it must *search* for the correct inference method to apply for every problem. Inflexible architectures do not require as much inference method search, since fewer methods can be specified. For very inflexible architectures such as TheoGT, *no* search is required. Hence, trading off flexibility for efficiency is a practical strategy for increasing architecture performance.

The problem with this efficiency-increasing approach is not the flexibility-efficiency tradeoff *per se* – it is the fact that the architecture forces the user or knowledge-base designer to operate at a fixed trade-off point. For instance, consider Theo and TheoGT. Assume a knowledge-base designer constructs a domain for which half its inferences require Theo's flexibility. The designer has only two options:

- Use Theo, and take coffee breaks.
- Use TheoGT after expending effort reformulating the domain.

I.e., the designer is constrained to operate at the flexibility-efficiency trade-off points defined by Theo and TheoGT.

A superior approach would be to offer the designer more options: allow him flexibility if he needs it, and increased performance if he doesn't – i.e., let the designer determine the flexibility-efficiency trade-off point via the characteristics of the domain. If this were possible, the designer's domain would operate correctly, and half of the designer's inferences would run efficiently – giving him the best of both worlds.

1.2 Thesis Goals

The previous section summarized two strategies for increasing architecture efficiency. Both of these strategies have problems. Speedup mechanisms may not be effective, and can actually decrease system performance because of the difficulty of managing speedup mechanisms. Reducing architecture flexibility constrains the user or knowledge-base designer.

This thesis seeks to increase architecture efficiency by resolving the problems associated with the previously discussed strategies. *Specifically, the thesis of this research is:*

- An architecture can autonomously and effectively manage multiple speedup mechanisms. In particular, the management scheme should be able to adapt to environment dynamics, and increase architecture performance over a wide range of situations.

- An architecture can automatically reduce flexibility-efficiency constraints, adapting to the requirements of the domain. That is, the architecture allow flexibility if needed, but trade off unnecessary flexibility for efficiency. Furthermore, the architecture should not require any interaction with the user or knowledge-base designer.

1.2.1 Speedup Mechanism Management Approach

Given that speedup mechanisms need a flexible management scheme to be effective, this thesis approaches the first goal by making the architecture itself responsible for managing its speedup mechanisms. The approach considered in this thesis is to embed an agent into the architecture. Its job is to continually monitor architecture operation, and invoke various speedup mechanisms when appropriate.

What are the advantages of this approach?

- The speedup mechanism management responsibilities are off-loaded from the user or knowledge-base designer.

- Because the agent can monitor architecture operation over the life of the system, the agent can modify any management decisions to handle environment dynamics. Hence, this approach is adaptive.

- The agent can monitor the architecture at a very fine-grained level. Therefore, the agent can make very fine-grained management decisions. Also, because the agent can monitor system behavior much more thoroughly than any human, it can do a better job of analyzing architecture operation and determining effective management schemes.

1.2.2 Reducing Efficiency-Flexibility Constraints

In general, architecture flexibility reduces efficiency because flexibility necessitates additional inference. This additional inference is needed guide the operation of the system, and is called *system-level* or *meta-level* inference. Hence, flexible architectures are less efficient than inflexible architectures because of the meta-level inference required. Moreover, higher levels of architecture flexibility require more extensive meta-level inference. However, if the domain has been constructed in such a way that a high level of architecture flexibility is not required, these extensive meta-level search paths are unsuccessful. The time spent exploring these paths is wasted.

Meta-level inference has properties that make it amenable to analysis prior to runtime. Using such a *static* analysis, it is possible to *prune* meta-level search paths. That is, unsuccessful search paths can be truncated at run-time, increasing architecture efficiency. Essentially, a static knowledge-base analysis defines a *meta-level inference boundary.* Search outside this boundary is known to fail and can be pruned.

How is pruning meta-level inference related to reducing efficiency-flexibility constraints? A domain requiring a high level of architecture flexibility results in a large inference boundary – not much meta-level inference can be pruned. On the other hand, domains that do not require fine-grained architecture configurability give a small inference boundary, allowing more search pruning and increasing architecture performance. Hence, this pruning technique essentially trades off unnecessary architecture flexibility for inference efficiency by examining the flexibility requirements of the domain.

1.3 Experimental Results Summary

The approach used in this thesis is to embed an agent into a learning architecture. The agent uses static an dynamic analyses to increase architecture efficiency. This agent is called ISM (Internal Speedup-Mechanism Manager). ISM is embedded into the Theo learning architecture. Theo is a very flexible architecture, and supports several speedup mechanisms. ISM's speedup mechanism management component manages three of them: caching, EBG, and Bounded-Cost EBG (BEBG), a variant of EBG. Experiments pitting ISM against Theo have been run on two real-world, natural domains with widely varying characteristics. In these experiments, ISM increases Theo efficiency by a factor of more than two on one domain, and a factor of twelve on the other.

1.4 Thesis Organization

Chapter 2 presents an overview of Theo and describes the speedup mechanisms supported by Theo that are relevant to this research.

In Chapter 3, a more precise description of the goals of this thesis is given, and an overview of the approaches used in this thesis are presented. Chapters 4 and 5 discuss in detail architecture-controlled speedup mechanism management and architecture efficiency-flexibility constraint reduction, respectively.

Chapter 6 discusses the overhead associated with automatic speedup mechanism management, and presents approaches for reducing this overhead.

Chapters 4, 5, and 6 each contain the results of very focussed experiments, demonstrating the effectiveness of each of the individual ideas and techniques of this thesis. In Chapter 7, the overall results of this work are presented.

Chapter 8 discusses related work. Chapter 9 summarizes the issues and results of the thesis, and considers some areas for future research.

Chapter 2

Theo Overview

As mentioned in Chapter 1, the approach taken in this research is to embed an agent into a learning architecture. This agent – called the Internal Speedup-Mechanism Manager (ISM) – uses static and dynamic (run-time) analyses to increase architecture efficiency. ISM uses the Theo learning architecture [Mitchell 91] as a testbed. Consequently, all performance data reported in this research has been determined using Theo. Although the ideas behind ISM are relevant to learning architectures in general, it is useful to have some familiarity with the details of Theo's operation to fully understand ISM. Furthermore, it is important to comprehend the speedup mechanisms managed by ISM: caching, EBG, and Bounded-Cost EBG. This enables an understanding of ISM's management decisions with regard to these mechanisms. In this chapter, the portions of Theo operation relevant to ISM are described, as well as caching, EBG, and Bounded-Cost EBG.

2.1 Theo Representation

Theo uses a *frame-based* representation. That is, objects are represented by *frames*, each of which can have properties or *slots* that describe the object. For instance, consider the following frame *box1*:

```
(box1 ...
  (height 5)
  (width *novalue*)
  (length *inferred.novalue*)
  ...
  )
```

In this frame, *height*, *width*, and *length* are all slots of the frame *box1*. Each slot may contain a value. In this knowledge-base fragment, the height of box1 equals 5.

In Theo, there are two slot values that have special meanings: **novalue** and **inferred.novalue**. If a slot value equals **novalue**, Theo does not currently know the value of that slot and Theo has not attempted to infer a value. If it contains **inferred.novalue**, Theo has attempted to infer the value for the slot but has found it impossible to infer to value.

Uniform Representation

One way to view frame-based representations is to think of values of slots of frames as corresponding to beliefs held about frames. For example, in the above knowledge-base fragment the system "believes" that the height of box1 equals 5. One of Theo's interesting features is its *uniform representation*. That is, in addition to holding beliefs about frames, Theo can hold meta-beliefs (beliefs about those beliefs), meta-meta-beliefs, *ad infinitum*. This means that slots can have subslots, subslots can have subsubslots, et cetera. Essentially, Theo can hold beliefs about *anything* in the knowledge-base.

To explain Theo's representation power more precisely, let us define recursively an *entity* as a frame or a slot of an entity. Since Theo represents beliefs as slots of entities, and Theo's uniform representation allows beliefs about any entities, any entity can contain slots. That is, slots can be nested. For instance, consider a knowledge-base fragment for the frame *box2*:

```
(box2 *novalue*
  (height 3
    (accuracy? high)
    ...
    )
  (width *novalue*)
  ...
  )
```

As in the previous knowledge-base fragment, Theo knows information about the height of the entity *box2*. Theo's representation, however, also allows it to hold information about the entity *box2 height*. In this case, this information concerns the accuracy of Theo's beliefs regarding the *height* of

box2. Although not included in the knowledge-base, *box2 height accuracy?* could contain further slots, et cetera.

Another notation can be used to show the information contained in the knowledge-base. Its general schema is: $(< entity >< slot >) =< value >$. So, (box2 height) = 3 and (box2 height accuracy?) = high. The list $(< entity >< slot >)$ is known as an address, query instance, or problem instance.

2.2 Theo Inference

Data can be stored and retrieved from the knowledge-base, making Theo useful as a kind of information repository. However, Theo is most useful as an inference engine. That is, in addition to simply looking up information in the knowledge-base, Theo can try to *calculate* or *infer* query instance values if they do not exist in the knowledge-base, based on Theo's operational semantics and other data in the knowledge-base. In this section, Theo's inference mechanisms are described.

2.2.1 Low-Level Inference

Suppose Theo must infer a value of the problem instance P (i.e., the solution to the problem P). Theo assumes that the knowledge-base contains a Lisp function F that can compute the value of P. Hence, to infer P, Theo locates F and applies F to P, using the value returned by F as the value of P. That is, $F(P) =< value >$.

At this level of abstraction, Theo is not directly responsible for determining the value of P; F is. However, it is Theo's responsibility to locate F. To explain how Theo does this, assume that P equals the address (s1 s2 ... sM sN). Since all addresses consist of an entity and a slot, P's entity equals (s1 s2 .. sM) and P's slot equals sN. Theo initially attempts to find F at the address (s1 s2 ... sN toget). If F does not exist in that location, Theo recursively *cdr*s down the address list, looking at (s2 ... sN toget), (s3 ... sN toget), et cetera, terminating when F is found or at (sN toget). Because Theo searches for F using the *toget* subslot of P, F is known as P's toget. If Theo cannot locate P's toget using the above strategy, Theo applies a default toget to the problem. This default toget resides at the (toget default.value) address of the knowledge-base.

As was previously mentioned, a problem instance consists of an entity and a slot – $P = (< entity >< slot >)$. The entity of P is known as

the *context* of the slot. Hence, this method of *cdr*ing down an address list searching for a value in the knowledge-base is called the *drop-context* inference method, since this method decreases the context of the problem instance. This is a way of searching for a value starting from a specific location, and moving to increasingly general locations. I.e., the more context the value of a slot has, the more specific its value is. If the slot value has less context, it is applicable to a larger problem class.

The above discussion shows that the knowledge-base designer can precisely specify Theo's behavior by inserting toget functions into the knowledge-base. Furthermore, the drop.context method of looking for a problem instance's toget gives the designer a fair amount of flexibility and control over how inferencing should be carried out. For instance if the designer wants a particular toget function to be generally applicable to a particular slot, it is stored in a location with little context. On the other hand, storing toget functions at locations with more context allows the designer to differentiate (to some degree) the inference behaviors of problem instances with identical slots.

2.2.2 High-Level Inference

Although the ability to specify Theo's inference behavior via toget functions gives the knowledge-base designer a high degree of flexibility and power, it is very inconvenient – the designer must specify Theo's behavior via low-level Lisp code. So, Theo provides a set of "built-in" inference mechanisms as well as a language for specifying inference patterns at a more abstract level than Lisp. In addition, Theo gives the knowledge-base designer a simple way of controlling these "higher-level" inference schemes.

Inference Methods

Theo's built-in inference mechanisms are known as inference *methods*. Some of the methods Theo supports are:

- Inheritance: use the generalization hierarchy to inherit the value of an address. For example, if P = (box1 height methods) and box is a generalization of box1, inheritance would try to infer P by querying (box height methods).

- Drop.context: described above. So, if P = (box1 height methods), drop.context tries (height methods).

- Default.value: determine the value of P by inferring the value of (P default.value). I.e., if P = (box height), default.value tries (box height default.value).

In addition to using these methods, the knowledge-base designer has the option of specifying inference patterns – essentially implementing new inference methods – using high-level Prolog-like rules. This gives the designer much more power and flexibility than the methods described above, yet is much easier than constructing low-level Lisp toget functions. Furthermore, EBG and BEBG, which are described later in this chapter, perform better with the more highly structured Prolog-like rules than with the arbitrary Lisp toget functions.

How can a knowledge-base designer actually use Theo's inference methods? Inference methods are specified for problem instances or classes by storing them in the *methods* subslot of the instance or class. The following knowledge-base fragment specifies methods for the problem instance (*box*1*height*) and for the problem class (*boxheight*).

```
(box1 ...
  (generalizations (box))
  (height 10
    (methods (inherits drop.context default.value)))
  )

(box ...
  (height ...
    (methods (inherits default.value)))
  )
```

Note that the drop.context and inheritance inference methods work by searching for values in progressively more "general" locations in the knowledge base. This gives the domain designer the ability to *configure* the architecture's operation – the architecture is flexible. For instance, the designer may wish to specify a single inference method for a large class of objects (say, the *area* class in Figure 1.1). If he later wishes to modify the methods for a small subclass (perhaps the *triangle* subclass in Figure 1.1) , he simply changes the methods of that subclass. The other elements of the class continue using the originally specified methods.

Implementation

As described earlier, if Theo cannot find a special-purpose toget for problem instance P, it uses a default toget. This toget essentially implements high-level inference by attempting to find the *methods* subslot of P, which contains a list of methods. Theo then proceeds by applying each method (in order) on P, until a value is returned. This value is taken to be the value of the problem instance.

Hence, a knowledge-base designer can specify which of Theo's predefined methods to use, and the order they should be used in, for any problem instance by specifying the *methods* of problem instances or problem classes. If the knowledge-base designer does not specify a set of methods values, Theo uses a default set.

2.2.3 Meta-Level Inference

One of the interesting features of Theo is that only Theo's low-level inference is "hard-coded." Theo's high-level inference behavior is specified entirely by data in the knowledge-base. For example, to enable the designer to use Theo-defined methods, a default toget function is stored in the address (toget default.value). Similarly, Theo's inference methods and their behaviors are actually defined in the knowledge base. Consequently Theo must continually "consult" the knowledge base to determine how it should operate. This kind of inference is called meta-level or system-level inference. It lowers Theo's efficiency, but increases its extensibility and flexibility.

Theo's meta-level inference overhead is exacerbated by the fact that many system-level slots can not be inferred directly – they require inferring other system-level slots. For instance, inferring the *methods* of a problem instance will almost always result in the query of a related slot: *fixedmethods*.

2.2.4 Prolog Rules

As discussed previously, one of Theo's methods is *prolog*. This method specifies that Theo can use *prolog rules* to attempt to infer the value of a query instance. For example, assume Theo must infer the *daughters* of *tom* – i.e., the query instance is (tom daughters). Further assume that the knowledge-base contains the following rule:

```
((daughters ?person ?d) :-
  (children ?person ?d)
```

```
(generalizations ?d female))
```

In order to understand how Theo uses prolog rules, some vocabulary must be defined. The above rule consists of three *literals*, one on each line. Theo's prolog rules are simple antecedent-consequent rules. The literals that follow the backwards-implication symbol :- are the *antecedents*; the literal that preceeds the :- symbol is the *consequent.* The antecedent of a rule P will be denoted A(P); the consequent will be denoted C(P). Each literal has the following form: $(< slot >< entity >< value >)$, and represents the following information: $(< entity >< slot >) =< value >$. Either the entity or value elements in any literal can be *variables.* Variables are preceeded by a ? symbol.

A rule's antecedent must be satisfied before the rule's consequent can be asserted. Hence, the goal of Theo's prolog rule interpreter is to determine a consistent set of bindings for the variables in a rule's antecedent that satisfies every literal in the antecedent. If this is possible, the rule's consequent can be asserted. However, since Theo is backward chaining, prolog rules are not applied until Theo is asked to infer a *relevant* query instance. A rule P is relevant to a query instance Q if C(P) *matches* Q. Such a match exists between Q and P if both the slot and entity of C(P) equal those of Q. This may or may not require binding some variables. In the above example, a match exists between the rule and (tom daughters) if *?person* is bound to the value *tom.*

Determining a set of consistent variable bindings for a rule's antecedent involves subgoaling on each literal of the antecedent. For example, in the above rule, the first antecedent literal is (children ?person ?d). During the matching process between the rule and query-instance, the prolog interpreter instantiated *?person* as *tom*, so the first antecedent literal has become (children tom ?d). To bind ?d, Theo constructs a recursive query instance (tom children), and assigns ?d to the value of this new query instance. The next literal in the rule's antecedent is (generalizations ?d female). Interpreting the previous literal has generated a set of bindings for ?d. However, the set of bindings must satisfy all the literals in the rule antecedent. This second antecedent literal denotes that the elements of ?d must also be female. Thus, Theo uses the second literal to filter out the initial instantiations of ?d that are not *female.* Hence, ?d equals the set of *female children* of *tom.* Since ?d equals the value of the rule consequent, the prolog interpreter returns the value of ?d as the result of the query instance.

2.3 Speedup Mechanisms

The research discussed in this thesis deals with managing speedup mechanisms. Because management strategies are specific to the characteristics of each speedup algorithm, it is important to have some understanding of each of the mechanisms that ISM manages. This section briefly describes each such mechanism: caching, Explanation-Based Generalization (EBG), and a variant of EBG called Bounded-Cost EBG (BEBG).

2.3.1 Caching

Caching is Theo's simplest and most used speedup mechanism. Caching the value of a query instance consists of memorizing that value – storing it in the knowledge-base. This is a very useful speedup mechanism if queries tend to be repeated. In Theo, this is very often the case. Many of Theo's query instances are system, or meta-level addresses, such as methods, fixedmethods, and toget addresses. Because values for these system-level addresses are typically initially stored in very general locations (i.e., addresses with little context) inferring them can require a large amount of search through the knowledge-base resulting in many intermediate query instances. If these values are cached in these intermediate locations, they can be immediately accessed in future queries, reducing inference time.

If the value V of a query instance P is cached, Theo also stores an *explanation* E of V in the *expl* subslot of P. The explanation describes how V was derived. More precisely, E is a list of addresses whose values were needed to infer P. For instance, consider the following:

```
(square1 ...
  (area 9
    (expl (((square1 area toget) (square1 area methods)
            (square area)))))
  )
```

This knowledge-base fragment shows that to infer the value for (square1 area), Theo used the following addresses: (square1 area toget), (square1 area methods), and (square area). If the values of these addresses were also cached, they would also have explanations. Hence, it is possible to build a directed graph giving a derivation of the inference result. Such a graph is called an *explanation structure*. One reason that explanations exist in Theo is that explanation structures are needed by the EBG algorithm.

Explanations are also used for truth maintenance. In the above sample explanation, the value of (square1 area) depends on the value of (square area). If the latter value changes, the former value is no longer valid, and should be erased, or uncached. Explanations provide a mechanism by which this kind of knowledge consistency may be maintained.

To decide which problem instances should be cached, for every problem instance P, Theo also infers (P whentocache). The value of this address determines Theo's caching strategy for every address. By default, Theo caches every problem instance.

2.3.2 Explanation-Based Generalization

Learning concepts from examples has been a primary research focus of machine learning. One such type of learning is *concept induction* – examining many examples of a concept to determine a general description for that concept. Explanation-based generalization, is another kind of algorithm. EBG generalizes from only a single concept instance by determining the *reasons* that the example is an instance of the concept to be learned. These reasons are given by the instance's *explanation*, and gives a set of sufficient conditions under which an example is an instance of the concept. Although this explanation is specific to the instance, the EBG algorithm generalizes the explanation, resulting in a more useful concept description. Unlike concept induction techniques, EBG is deductive – its concept generalizations are justified.

The input/output behavior of EBG is as follows. Given:

- Goal Concept: the concept definition describing the concept to be learned.
- Training Example: an example of the goal concept.
- Domain Theory: a theory explaining why the training example is an instance of the goal concept.
- Operationality Criterion: a predicate over concept definitions, specifying the form in which the learned description must be expressed.

EBG determines:

- A generalization of the training example that is a sufficient definition for the goal concept (and is hence more specific than the goal concept), and that satisfies the operationality criterion.

The goal concept is a high-level concept description. When EBG is applied, it uses the goal concept and the domain theory to explain why the training example is an instance of the concept to be learned. The operationality criterion determines the point at which the explanation terminates. At this point, EBG has constructed an *explanation structure* for the training example. To generalize this structure, EBG simply regresses the goal concept through the explanation structure.

From the description of this algorithm, it is clear that EBG is effective when applied to problems with similar explanation structures. In effect, EBG produces a "macro" which describes a high-level concept directly in low-level terms, eliminating the intermediate-level computation that would otherwise result.

In Theo, EBG generates prolog rules. Thus, the rule interpretation process used for user and system-defined rules is identical to the rules generated by EBG.

2.3.3 Bounded-Cost EBG

One of the problems with EBG is that it can generate rules whose form causes the rule to be very costly to apply. For instance, consider the following rule:

```
((child ?p ?c) :-
  (parent ?c ?p))
```

Assume this rule were to be applied to the query instance (tom child). Consequently, ?p is bound to tom. To instantiate ?c, Theo uses the antecedent literal (parent ?c tom). This literal has a different form than those discussed previously. Specifically, its *value* is bound and its *entity* is unbound. To bind an entity whose slot S and value V are known, Theo must search through all entities E in the knowledge-base that are elements in the domain of the slot, and calculate the value of (E S). Those with values that equal V are collected, and bound to the literal's entity variable. Hence, binding variables in each literal of a rule's antecedent requires $|Domain(slot)|$ recursive Theo queries, where $|Domain(slot)|$ equals the size of the domain of slot.

Contrast this situation with the following rule:

```
((A ?x ?v) :-
   (B ?x ?i)
   (C ?i ?v))
```

Here, as Theo attempts to bind the variables of each literal in the rule's antecedent, only the *value* variables are unbound, rather than the *entity* variables. Binding a *value* variable of a literal requires only one recursive Theo inference.

The former type of rule is known as an unbounded-cost, or "expensive" rule; the latter rule has bounded cost. The variables that cause a rule to have unbounded cost are known as expensive variables. Note that there are different levels of complexity: the above rule has only one expensive variable; the following rule has two:

```
((A ?x ?v) :-
  (B ?i ?x)
  (C ?j ?i)
  (D ?j ?v))
```

The number of expensive variables in a rule gives the rule's "degree of expensiveness."

In some situations, bounded-cost rules are preferable. The next section discusses a variant of the EBG algorithm which generates only bounded cost rules, called bounded-cost EBG or BEBG.

The BEBG Algorithm

BEBG is identical to EBG, but it adds a postprocessor. This postprocessor

- determines which variables cause the rule to have unbounded cost
- instantiates these variables to the values used in the training example.

Although variable instantiation transforms unbounded-cost rules into bounded-cost rules, it drastically reduces the rule's generality. For example, the rule

```
((child ?p ?c) :-
  (parent ?c ?p))
```

becomes bounded-cost if we bind ?c to a value:

```
((child ?p kim) :-
  (parent kim ?p))
```

On the other hand, this rule has become so specific that it may not be useful. Hence, BEBG tends to produce rules that are more efficient, but less general, than EBG.

Determining the variables of a rule that cause it to be unbounded-cost can be problematic. For example, consider this rule:

```
((A ?x ?y) :-
  (B ?i ?x)
  (C ?j ?x)
  (D ?i ?j)
  (E ?j ?i))
```

In its current form, this rule is expensive. Its cost can be bounded, however, by instantiating either ?i or ?j. How does BEBG decide? BEBG instantiates variables using a "greedy" algorithm. That is, it constructs a list of all possible expensive variables, and instantiates the single variable that minimizes the degree of expensiveness of the rule. This heuristic is repeated until the rule has bounded cost.

Chapter 3

ISM Overview

This thesis examines some ways in which the efficiency of a learning architecture can be increased. A system called the Internal Speedup-Mechanism Manager (ISM) has been implemented to study some classes of architecture speedup techniques. ISM is built on top of Theo, and increases Theo efficiency using both dynamic (at run-time) and static (pre-run-time) strategies. This chapter discusses ISM's goals and presents an overview of ISM's speedup techniques.

3.1 ISM Performance Goals

The goal of ISM is to increase Theo's efficiency. However, ISM's behavior cannot be unconstrained. There must be limits to how ISM affects Theo's operation. Furthermore, there are many metrics by which efficiency increases can be judged. What, precisely, are ISM's goals, and how can ISM be judged? This section reviews the system constraints and speedup goals taken in this research.

3.1.1 ISM Constraints

In this thesis, ISM is constrained to *preserve Theo's behavior*. That is, from the user's point of view, ISM cannot modify:

- Theo's inference capabilities – Speeding up the system should not decrease Theo's inference ability. Efficiency gains should not affect architecture competence.

- Theo "correctness" – Increasing efficiency should not change Theo's "input-output" behavior. That is, a system with ISM and a system without ISM, if given the same knowledge bases and queries, should return identical results.

From the knowledge-base designer's point of view, ISM should not change:

- Theo programmability – Theo should retain its flexibility and configurability. The ability of the user or knowledge-base designer to specify Theo's inference behavior at a fine-grained level should be maintained.

Finally, ISM should not require additional:

- User–Theo or designer–Theo interaction – The techniques used by ISM should not require any user or designer input, involvement, or expertise. ISM should operate autonomously.

Essentially, ISM should modify Theo only to the extent that efficiency is enhanced. That is, users and knowledge-base designers should not have to be aware of ISM's presence.

Note that these constraints may not always be appropriate for all situations. For instance, if in a certain domain Theo is used merely to assist the user in some real-time task, speed may be more important to the user than inference competence. That is, the user may wish to trade some inference ability to decrease response time. Similarly, under some situations a user might be willing to give up correctness for speed. User's of the Calendar Learning Apprentice domain, for example, might trade a 5% decrease in correctness for a 50% increase in efficiency.

Nevertheless, to ensure ISM's applicability across a wide range of situations, ISM is constrained to preserve Theo's functionality.

3.1.2 ISM Speedup Goals

Assuming that ISM preserves Theo's behavior, the purpose of ISM is to increase Theo's efficiency. Although there are different ways to judge ISM's competence at this task, perhaps the most natural measure of ISM effectiveness is: *how much does ISM speed up Theo's inference over the lifetime of the system?* Given this measure, ISM's goal is to minimize architecture execution time over the life of the system.

Unfortunately, there are some problems with this goal. First, this goal can result in inference cost patterns that might be inappropriate for some

tasks. For instance, consider an interactive task. In such a setting, users are generally more concerned with individual query response times than with overall system efficiency. If ISM's goal is to minimize execution time over the life of the system, Theo might execute 90% of the user's queries in only .1 seconds, and 10% of the queries in 1.1 seconds, resulting in an average of .2 seconds/query. Users might prefer that *all* queries be inferred within .25 seconds, even though total system efficiency would decrease.

Another problem with this goal is that it is not specific with respect to exactly what speedup strategies ISM should use. It is difficult to implement a system that minimizes overall execution time because such a system must consider an immense number of possible strategies that could speed up the system, and it is most likely impossible to determine the optimal strategy.

If maximizing overall system efficiency is unsuitable as a goal, what is an appropriate alternative? For this research, ISM's primary goal is defined from a practical perspective. ISM's purpose is to be useful for all users, in all situations. That is, ISM is successful if there is never an instance where a user prefers to use Theo *without* ISM. How can ISM meet this goal? By ensuring that *ISM increases Theo's efficiency for every query instance*. Note that this goal does not attempt to ensure optimality. Rather, it ensures that ISM will never decrease system efficiency. If ISM is successful according to this criterion, ISM will benefit all users with all kinds of usage patterns – its speedup strategies will be universally useful. Hence, this *per query* speedup goal does not suffer from the problems of the *system lifetime* speedup goal.

Note that this goal is an *ideal*; it is impossible for any management strategy to guarantee performance increase in all situations, because one can construct situations for which Theo's default operation is already optimal. Hence, although ISM strives for the ideal, it cannot make any performance guarantees.

3.2 ISM Goal Consequences

ISM's goal has some interesting ramifications with regard to its necessary characteristics and its power. These issues are discussed in this section.

3.2.1 Adaptivity

Consider the following scenario: Theo begins operation on a domain with stable characteristics, for which Theo's default operation is not well-suited. ISM finds and implements a speedup strategy that is superior. Suddenly, the

characteristics change. Now, Theo's default operation is superior to ISM's strategy. For ISM to fulfill its performance goal, it must find a new strategy.

This scenario demonstrates that ISM must be *adaptive*. No single speedup strategy can be superior to Theo's default operation in all situations. To guarantee its speedup goals, ISM's strategies must reflect changes in the architecture environment. Examples of changes include query distribution shifts and shifts in the stability of knowledge base data. Adaptivity is an important second-order goal, implied by ISM's primary performance goal.

Adaptivity implies another performance goal: ISM must outperform any *fixed* speedup mechanism management strategy over a range of domains with varying characteristics. An example of a fixed speedup mechanism management strategy is Theo's default strategy of handling caching and EBG: cache at every opportunity, and never invoke EBG or BEBG. This strategy may be in fact optimal for some domains. However, the adaptivity goal is meant to ensure that ISM is, *in general*, superior to any single speedup strategy.

3.2.2 Goal-Induced ISM Limitations

ISM's per-query speedup goal tends to limit ISM's scope and power. Ensuring that architecture performance will never decrease is a strong constraint, and results in the following limitations:

- No run-time experimentation. ISM simply does not have time to experiment to determine the most effect speedup mechanism management schemes. This may result in less than optimal decisions when sensor data is incomplete. Furthermore, this constraint can lead to "quick and dirty" management decisions, since a more careful analysis of the architecture's operation may be prohibitively expensive.

- Very conservative behavior. ISM cannot be very speculative – it must be fairly certain of the speedup potential of a particular management decision before that decision is adopted. This risk-averse behavior means ISM tends to make speedup gains incrementally. Dramatic speedup gains can be relatively rare.

- No learning techniques employed. ISM's opportunities to learn how to properly manage speedup mechanisms are limited due to the overhead of learning. ISM simply cannot afford the time needed to, say, learn how to classify the query instances that should be cached. It

is important to note although ISM could learn off-line, this strategy could have problems due to changes to the architecture environment occurring at run-time. This kind of situation requires on-line learning.

- Possibly skewed management priorities. ISM must ensure that under no situations does its operation slow Theo down. Thus, ISM may have to trade off potential speedup in one situation to make sure there is no slowdown in another. This is a problem if the speedup potential is much larger than the slowdown potential, or if the speedup situation is much more common than the slowdown situation. One way in which this tradeoff can occur is through sensing. The sensors needed to detect the potential speedup situation may be too expensive in some other situations. If this is the case, even if the potential speedup is large, ISM cannot use these sensors, and hence the speedup opportunity is lost.

ISM's goal of preserving Theo's behavior is also limiting. Many of these limitations are obvious, but some of the more interesting issues to consider are:

- Correctness behavior. It is possible that ISM could speed Theo up dramatically at only a small accuracy cost. For instance, deleting some infrequently used *methods* could have such an effect. Unfortunately, ISM is not given the opportunity for this kind of optimization.

- Theo specification. Theo is implemented in such a way that its operation can be configured via the knowledge base at a very fine-grained level. The cost of this flexibility is inference overhead; however, much of this flexibility is never used. Configurability could be sacrificed for efficiency. However, this kind of optimization is not allowed.

3.3 ISM Approach

How does ISM increase Theo's efficiency? ISM utilities two strategies: a "dynamic" strategy that relies on run-time analyses of Theo's operation to improve system performance, and a "static" strategy that uses a pre-run-time knowledge-base analysis to tune Theo's inference. These strategies are introduced in this section.

3.3.1 Dynamic Strategy

ISM's dynamic component uses speedup mechanism management to increase Theo's performance. That is, while Theo is operating, ISM considers (for each query instance) whether caching, EBG, or BEBG can increase Theo's efficiency, and invokes these mechanisms appropriately. Because Theo's inference is highly recursive, each top-level inference can result in hundreds of query instances. Hence, ISM faces quite a task.

To give ISM the ability to analyze Theo's operation and manage Theo's speedup mechanisms at run-time, ISM is designed as an agent embedded into the Theo architecture. As such, ISM is comprised of sensors, decision criteria, and effectors. The sensors monitor Theo's operation, collecting the data needed to make appropriate speedup mechanism management decisions. The effectors actually apply the various speedup mechanisms. The decision criteria map sensing information to appropriate speedup mechanism management decisions.

ISM's decisions are based on a utility analysis. During the run-time analysis of a query instance, ISM calculates the utility of every speedup mechanism on the instance. This gives the expected time savings resulting from the application of the speedup mechanism. Using this data, ISM determines which mechanisms to apply to a particular query instance. The details of ISM's utility analysis, and ISM's strategy for determining which speedup mechanisms to apply given this information, are discussed in the "ISM Run-Time Optimizations" chapter of this thesis.

One of the main problems faced by an architecture such as ISM is efficiency. Monitoring Theo's operation and calculating speedup mechanism utilities can be very expensive. If ISM must make speedup mechanism management decisions for every query instance, this ISM overhead cost can be prohibitive. In fact, for good performance, it is necessary to implement strategies to reduce ISM overhead. ISM incorporates two such strategies, which are described in the "Managing ISM Overhead" chapter.

Effects of ISM Goals on Speedup Mechanism Management

Recall that ISM's primary goal is to out-perform Theo on a per-query basis. Because Theo never applies EBG or BEBG, ISM can be very conservative in utilizing these mechanisms. This observation has important ramifications for reducing ISM overhead. Essentially, ISM need not be very complete in analyzing when EBG and BEBG are useful – it does not need to consider the

effects of EBG and BEBG on every query instance. This issue is discussed in Chapter 6.

3.3.2 Static Strategy

In compiler design, a static code analysis can reveal various types of optimizations. ISM's static speedup strategy is based on the same general idea.

Inference in Theo consists of a search through the knowledge base to find information that is needed to determine the value of the query instance. Often, this search is inefficient. Unsuccessful paths are explored. Because Theo's operation is specified by values in the knowledge base, in some cases it is possible to roughly determine Theo's search paths for classes of query instances from a static analysis of Theo's knowledge base. Furthermore, by examining the initial state of the knowledge base, it is possible to determine which search paths *cannot* be successful. ISM performs such an analysis, and at run-time, uses this information to *prune* Theo's inferencing.

This technique is discussed in the Chapter 5.

3.4 General Issues

The main thrust of this thesis is the investigation of speedup mechanism management techniques to increase architecture efficiency. Thus, the main purpose of this thesis is to describe one such successful system – essentially, to relate the sensors, effectors, and decision criteria of ISM. In addition to confronting the actual management issues, however, this research also attempts to address some broader, more general concerns. These include:

- Which speedup mechanisms are most useful when managed by an ISM-like mechanism?
- What are the characteristics of these speedup mechanisms?
- Under what kinds of domains is an ISM-like system most effective?
- What features in the learning architecture are needed to support ISM?

Obviously, there are ways of increasing architecture efficiency that are independent of speedup mechanisms. Since the wider goal of this work is to speed up architectures – Theo in particular – a part of this research addresses

a way to speed up inference that does not involve speedup mechanisms: ISM's static analysis technique. This portion of the thesis gives one method by which Theo inference can be optimized with no modifications to Theo behavior. Some of the wider issues involved with this technique include:

- What are the characteristics of a system's inference mechanism that would make this kind of optimization useful?
- What are the advantages of this kind of "inefficient" behavior?
- What features in the learning architecture are needed for this kind of optimization to be effective?
- Under what situations is this optimization useful?

3.5 Summary

In this chapter, ISM's goals were defined, and the effects of these goals on ISM's operation were considered. ISM's two speedup approaches were introduced, and some of the broader issues that this thesis addresses were overviewed.

Chapter 4

Managing Speedup Mechanisms

This chapter discusses the details of ISM's run-time optimization strategy, which manages the following three speedup mechanisms: caching, EBG, and Bounded-Cost EBG (BEBG). The aspects of these learning algorithms that are relevant to this research have been described in Chapter 2.

ISM's speedup mechanism management strategy relies on estimating the expected speedup, or utility, gained from applying each mechanism for each situation faced by the architecture. ISM bases its actions on these utility measures. The first section in this chapter is devoted to describing how ISM estimates speedup mechanism utility. The rest of the chapter explains the management choices available to ISM, the decision criteria on which ISM bases its management choices and actions, and the relation between these actions and ISM's utility estimates.

4.1 Speedup Mechanism Utility

This section describes and analyzes ISM's utility estimates for each speedup mechanism. To show how these estimates have been derived, a formulation of each mechanism's "ideal" utility is first presented, assuming perfect information about both past and future events. Since ISM unfortunately does not have access to perfect information – either due to sensor limitations or to a lack of knowledge about future events – ISM must estimate unavailable information. These estimates degrade the ideal utility formulations to utility *approximations*. ISM's approximations are discussed and justified. The

sensors needed to determine the information relevant to the utility approximations are then presented, followed by a description of the effectors needed by ISM to control each speedup mechanism.

Since the focus of this thesis is architecture speedup, the relevant measure of utility in this research is time – the amount of time a speedup mechanism, if invoked, saves Theo. Any speedup mechanism management decision can effect performance over the lifetime of the system. Hence, the most natural timeframe over which speedup mechanism utilities could be measured is system lifetime. However, it is more convenient to calculate utilities over smaller, finite periods of time. In the analyses presented in this chapter, the speedup mechanism utilities of a query instance Q are computed between successive queries to Q. That is, the *utility calculation time-frame* is taken to start when Theo begins to infer Q, and end with the first subsequent time Q is requeried.

Sensing Tradeoffs

Because ISM cannot sense future events, it relies on monitoring the past to predict the future. For instance, ISM assumes that it can use query distribution *histories* to estimate future distributions. One problem in designing sensors of this sort is determining the history "window" of the sensor – the amount of past information kept by the sensor. There is a tradeoff associated with sensor window-size: accuracy verses reactivity. Typically, larger window sizes result in potentially more accurate sensors, due to the larger amount of information available. On the other hand, large sensor windows mean the sensor is typically poor at detecting any dynamics in the information being monitored (such as query distributions), since these dynamics may be initially misinterpreted as noise.

Many of ISM's sensors require making tradeoffs of this kind. The most important issues in determining a reasonable tradeoff point are

- accuracy verses reactivity
- sensor efficiency

One of the important issues in designing an ISM-like mechanism is deciding how to make these tradeoffs. ISM has different tradeoff points for different sensors. The reasons why ISM chooses a particular sensing design with respect to these criteria are important to keep in mind as the sensors are described.

4.1.1 Caching

Ideal Utility

Consider the ideal utility of caching. Caching the result of a query instance Q is useful when:

- Q is subsequently requeried, and
- the values in the knowledge base that Q depends on (i.e., Q's *contributors*) do not change until Q is requeried at least once.

In this situation, the benefits of caching a query instance equals the inference time of the instance's subsequent requery.

On the other hand, caching Q always incurs the following costs:

- time spent creating or modifying the dependents slots for each of Q's contributors to include Q
- time spent inserting the value and explanation of Q into the knowledge base
- space needed to store the value, explanation, and dependents of Q

Furthermore, when the value of the query instance Q is cached, and Q's contributors change, Theo's truth-maintenance system causes the following elements in the knowledge base to be modified or deleted:

- Q's value
- Q's explanation
- the dependents slots of Q's contributors

To analyze the ideal utility of caching more formally, let us define the following quantities:

$U_{cache}(A)$ = the marginal utility of caching query instance A.
$\mathcal{R}(A)$ = the inference result of query instance A.
$\mathcal{E}(A)$ = the immediate explanation for A – i.e., the *expl* slot of A.
$\mathcal{D}(A)$ = the immediate dependents of A.
$\mathcal{L}(l)$ = the length of list l.
$\tau(n)$ = the inference time of the nth future query instance.
$\eta(A)$ = an integer n representing the nth future query

instance such that that query instance equals A and
it is the first such instance.

$$\sigma(A,n) = \begin{cases} 1 & \text{if the contributors of } A \text{ are stable for the next } n \text{ queries} \\ 0 & \text{otherwise} \end{cases}$$

$$\sigma^{-1}(A,n) = \begin{cases} 0 & \text{if the contributors of } A \text{ are stable for the next } n \text{ queries} \\ 1 & \text{otherwise} \end{cases}$$

K_{put} = time required for Theo to store 1 value in the KB

K_{cons} = seconds/cons-cell: how much space "costs" in terms of time

Note that some of these quantities are unknown. Calculating the precise utility of a speedup mechanism, for example, requires knowledge of future events. However, these quantities allow us to define precisely the utility of caching. Then we will consider how ISM approximates its utility calculations by estimating some of these unknown quantities.

Ideal Utility Calculations

The overall utility of caching query instance A equals the benefits of caching A minus the space and time costs of caching. That is,

$$U_{cache}(A) = Benefit_{cache}(A) - K_{cons}Cost^{space}_{cache}(A) - Cost^{time}_{cache}(A) \quad (4.1)$$

The amount of time caching saves equals the time it takes to infer the problem instance when it is requeried. However, this is true only when the query instance's contributors are stable. Hence,

$$Benefit_{cache}(A) = \sigma(A, \eta(A)) \times \tau(\eta(A)) \quad (4.2)$$

Now, consider the time costs of caching. There is an initial time cost as well as a TMS time cost associated with caching. The initial time cost results from the time it takes to store data in the knowledge base. This data consists of the value of the query instance, the query instance's *expl* slot, and the *dependents* slot of each element of the query instance's explanation.

$$Cost^{time}_{cache}(A) = Cost^{time}_{initial}(A) + Cost^{time}_{TMS}(A) \quad (4.3)$$

$$Cost^{time}_{initial}(A) = K_{put} \times (\mathcal{L}[\mathcal{E}(A)] + 2) \quad (4.4)$$

TMS time cost only manifests if the contributors of the query instance are unstable, since only in this situation must values in the KB be deleted. There

are two cases to consider. When the query instance is a top-level query, an unstable contributor results in the deletion of the value of the query instance, the *expl* slot of the query instance, and the removal of the query instance from the *dependents* slot of each member of the explanation of the query instance. If the query instance is not a top-level query, the marginal additional work done by the TMS due to caching the query instance amounts to only the deletions of the value, explanation, and dependents of the query instance.

$$Cost^{time}_{TMS}(A) = \begin{cases} \sigma^{-1}(A, \eta(A)) \times K_{put} \times (\mathcal{L}[\mathcal{E}(A)] + 1) & \text{if } A \text{ is top-level} \\ \sigma^{-1}(A, \eta(A)) \times 3K_{put} & \text{otherwise} \end{cases} \tag{4.5}$$

Finally, the space cost of caching the query instance equals the total amount of KB space needed by the caching mechanism multiplied by the amount of time that this space is being used. This space cost is "translated" into a time cost via K_{cons}.

$$Cost^{space}_{cache}(A) = \eta(A) \times (\mathcal{L}[\mathcal{R}(A)] + \mathcal{L}[\mathcal{E}(A)] + \mathcal{L}[\mathcal{D}(A)]) \tag{4.6}$$

Caching Utility Unknowns

In the above analysis, the following quantities are unknown:

- $\sigma(A, \eta(A))$
- $\tau(\eta(A))$
- K_{cons}

The first two items involve future events. The last item can be calculated, in principle. To do so, one could initiate two identical Theo runs with identical starting states, with one exception: add to one of the starting states a fixed data structure of known size. Theoretically, the space taken up by this structure should slow one of the runs down, due to effects such as garbage-collection and disk swapping. K_{cons} can be estimated as:

$$\frac{TimeDiscrepancy}{DataStructureSize \times NumberOfInferences} \tag{4.7}$$

In practice, this number very small small – virtually 0.

Estimating Caching Utility

ISM must estimate the values of these unknowns to use the utility calculations given above. ISM's estimates are:

$$\sigma(A, \eta(A)) \approx Prob_{stable}(A) \quad (4.8)$$

$$\tau(\eta(A)) \approx Cost_{inf}(A) \quad (4.9)$$

$$K_{cons} \approx 0 \quad (4.10)$$

Calculating $\sigma(A, \eta(A))$ requires knowledge of future events, which ISM does not have. ISM estimates this boolean quantity by the probability that A will remain stable before it is requeried: $Prob_{stable}(A)$. Also, rather than tracking the stability of the query instance's contributors directly, ISM does so indirectly by tracking the value of the query instance itself – a much cheaper alternative. That is, ISM keeps a history of past answers for the query instance. This history reveals when the query instance's value has been stable and unstable, and can be used to calculate the probability of stability of the query instance. Note that the stability of the problem instance value is not equivalent to the stability of the values of the problem instance's contributors. For instance, assume Theo needs to infer (box area), and does so by multiplying (box height) by (box width). If (box height) = (box width) = 2, then (box area) = 4. Suppose that the value of (box height) is changed to 4, and (box width) to 1. (box area) still equals 4. Because the contributors of (box area) have changed, (box area) is unstable. However, the value history of (box area) has remained stable. In this case, if ISM used the value history of the problem instance to conclude that the values of the instance's contributors were stable, ISM would be incorrect. This is known as the *sensor aliasing* problem, and will be discussed further in the Experimental Results chapter.

To estimate $\tau(\eta(A))$, ISM assumes that the time needed to infer a query instance in the future equals the time needed to infer the query instance at present, $Cost_{inf}(A)$. Note that $Cost_{inf}(A)$ represents true inference time – time required if A were not cached. Given ISM's lack of knowledge concerning future events, this is a plausible estimate, and very efficient. This approximation tends to be very poor during Theo's first few queries, since Theo tends to speed up dramatically during this time (mainly due to speedup from caching system slots). This estimate becomes much more accurate after an initial "warm-up" period, however.

The third approximation shows that ISM essentially disregards space costs in its utility calculations. Space costs are minor enough to be ignored

when making caching decisions. However, ISM does *indirectly* minimize space costs by considering speedup mechanism application on only a subset of Theo's query instances. Essentially, speedup mechanisms are only useful for query instances that satisfy a set of *usefulness criteria*. ISM's sensors make initial assumptions that can cause ISM to apply speedup mechanisms to query instances that do not satisfy these criteria. By using these criteria to filter the query instances that ISM considers, the space used by the speedup mechanisms is significantly decreased. Hence, ISM does not explicitly consider space costs in its utility calculations, but does so implicitly via the usefulness criteria. This will be explained in detail in Chapter 6.

ISM's estimates lead to the following expression for caching utility:

If A is a top-level query instance:

$$\begin{aligned} U_{cache}(A) &= Prob_{stable}(A)Cost_{inf}(A) \\ &\quad -(K_{put}(\mathcal{L}[\mathcal{E}(A)]+2)+K_{put}(\mathcal{L}[\mathcal{E}(A)]+1) \end{aligned} \tag{4.11}$$

If A is not a top-level query instance:

$$\begin{aligned} U_{cache}(A) &= Prob_{stable}(A)Cost_{inf}(A) \\ &\quad -(K_{put}(\mathcal{L}[\mathcal{E}(A)]+2)+3Prob_{unstable}(A)K_{put} \end{aligned} \tag{4.12}$$

Caching, Not Caching, and Uncaching

ISM uses its caching utility estimates to decide whether or not to cache a currently uncached query instance. If $U_{cache}(A) > 0$, caching results in speedup, and the value of A is cached. If $U_{cache}(A) < 0$, caching results in slowdown, and the value of A is not cached.

However, assume that ISM had previously decided to cache a query instance. How can ISM subsequently decide that the query instance should be uncached? Since the costs of caching do not equal the costs of uncaching, the utilities of caching and uncaching should be different. ISM is implemented in such a way that it makes management decisions only for query instances that do not have cached values – independent of the instances' caching strategy. Because ISM is applied only under this situation, ISM handles uncaching the same way it handles caching: if $U_{cache}(A) < 0$, A is uncached. That is, $U_{cache}(A) = -U_{uncache}(A)$.

ISM determines its caching strategy for a query instance at the conclusion of that instance's inference. This allows ISM to use the sensing information gained during the inference itself.

Caching Utility Sensors

The above utility analysis/approximation reveals that the following information is used by ISM to estimate caching utility:

- $Prob_{stable}(A)$: probability that the value of query instance A does not change before A is requeried
- $Cost_{inf}(A)$: time needed to infer A
- K_{put}
- $\mathcal{L}[\mathcal{E}(A)]$

$Prob_{stable}(A)$ is monitored by keeping a history of the results of A, from which the probability of stability is calculated. This kind of sensor must balance accuracy and reactivity, a tradeoff that has been discussed previously. ISM implements this sensor by storing N most recent answers to A in the knowledge base as a subslot of A, and uses this information to calculate $Prob_{stable}(A)$ on demand. Since leverage from managing caching and uncaching derives from the manager's ability to respond to environmental dynamics quickly (by applying the appropriate alternative), this data is monitored using a relatively short-term sensor, with $N = 5$. A short-term sensor is also more efficient in this case, since less data needs to be stored, and calculating $Prob_{stable}(A)$ is linear in N. For a novel query instance, this sensor has no stored data from which probabilities can be calculated. In this situation, $Prob_{stable}(A)$ is taken to equal 1.

$Cost_{inf}(A)$ is implemented by directly timing Theo's operation as it attempts to infer A. This resulting time – and only the most recent time – is stored in the knowledge base. I.e., this is a very short-term sensor. However, this allows ISM to immediately observe any environmental dynamics causing changes in inference times, or speedup effects resulting from a management action. Also, a short-term sensor operates more efficiently.

K_{put} is measured prior to run time. To determine this value, a function that inserts a value into the knowledge base is simply applied and timed.

$\mathcal{L}[\mathcal{E}(A)]$ is monitored directly. When Theo returns the value of a query instance, its explanation is also returned. This allows ISM to calculate the number of elements in the explanation.

4.1.2 EBG

Factors Influencing EBG Utility

It is useful to consider the similarities and differences between caching and EBG. Obviously, both can reduce inference time. The ways in which Theo uses the results of these speedup mechanisms are quite different, however. Caching results in a memorized fact. EBG, on the other hand, essentially results in an "inference shortcut" which Theo applies to find answers to query instances. As a consequence, from the architectural viewpoint, EBG is a simpler speedup mechanism than caching. Caching needs a truth-maintenance system to maintain the correctness of the knowledge base. EBG does not require such architectural support.

Because EBG learns an inference method rather than a simple fact, its results tends to be much more generally applicable than the results of caching. EBG learns rules that tend to be applicable to sets of query instances, rather than to a single instance. Furthermore, these rules do not require the stability of query instance contributors for usefulness – although they do need the knowledge base domain theory to be stable, which is virtually always true. Unfortunately, EBG has a higher overhead than caching. EBG overhead comes from the following factors. Note that EBG is considered *in isolation*. The effects of other speedup mechanisms such as caching are factored out.

- Rule generation. Applying the EBG algorithm can be expensive.
- Rule application. As discussed in Chapter 2, learned rules may be expensive – meaning the time it takes to apply a rule is at least linear in the number of objects in the knowledge base. Even if a learned rule is not expensive, rules take time to apply, since every clause in the rule results in a Theo query instance.

Essentially, there is a generality-cost tradeoff between caching and EBG. Caching has very little overhead, but is useful in only a small number of situations. EBG is generally more useful, but using learned rules is less efficient than using cached knowledge.

Ideal Utility

In addition to the quantities defined for caching, to determine the ideal utility of EBG, let us define:

$|domain(R,v)|$ = the number of unique instantiations of expensive variable v in rule R (i.e., the size of the domain associated with v).

$U_{EBG}(A)$ = the utility of applying EBG to query instance A.

$Ebg(A)$ = the rule resulting from applying EBG to A.

$RSucc(R,A) = \begin{cases} 1 & \text{if rule } R \text{ is successfully applied to query instance } A \\ 0 & \text{otherwise} \end{cases}$

$\mathcal{E}^*(A)$ = the complete explanation of query instance A.

$\mathcal{Q}(i)$ = the ith future query instance.

$Ants(R)$ = the antecedents of rule R.

$ExpVars(R)$ = the expensive variables of rule R.

$\alpha(R,A) = \begin{cases} 1 & \text{if } R \text{ is applicable to } A, \text{ i.e., if } slot(A) = car(head(R)) \\ 0 & \text{otherwise} \end{cases}$

$\hat{\alpha}(R,A) = \begin{cases} 1 & \text{if } R \text{ is applied to } A \\ 0 & \text{otherwise} \end{cases}$

K_{EBG} = seconds/length-of-explanation: query instance explanation length to time cost of applying EBG to query instance

The utility of applying EBG to query instance A equals the difference between benefits of EBG and the costs of EBG. Note that these costs and benefits are measured over the same period as those of caching.

$$U_{EBG}(A) = Benefit_{EBG}(A) - Cost_{EBG}(A) \tag{4.13}$$

The amount of time saved by a rule learned using EBG depends on the number of query instances on which the rule can successfully be applied during the given period, and the time Theo would have taken to infer each of these without using speedup mechanisms.

$$Benefit_{EBG}(A) = \sum_{i=1}^{\eta(A)} [RSucc(Ebg(A), \mathcal{Q}(i)) \times \tau(i)] \tag{4.14}$$

There are several costs associated with EBG: space cost, the initial time cost needed to execute the EBG algorithm, and the marginal time cost needed to apply rules learned using EBG.

$$Cost_{EBG}(A) = Cost^{time}_{initial}(A) + Cost^{time}_{marginal}(A) + Cost^{space}(A) \tag{4.15}$$

The space cost of EBG is proportional to the amount of KB space needed to store the learned rule multiplied by the amount of time that this space is

being used. As is the case with caching, this space cost is "translated" into a time cost via K_{cons}.

$$Cost^{space}(A) = K_{cons} \times \mathcal{L}(Ebg(A)) \times \eta(A) \tag{4.16}$$

The EBG algorithm constructs the explanation structure of the query instance and then regresses the goal concept through this structure. Hence, its execution time is linear in the size of the explanation structure. EBG's initial cost consists of this execution cost plus cost of inserting the rule into the KB.

$$Cost^{time}_{initial}(A) = K_{put} + K_{EBG} \times \mathcal{L}(\mathcal{E}^*(A)) \tag{4.17}$$

EBG's per-query (or marginal) cost for some period of time depends on the number of times (during the period) that the rule is applied multiplied by the time it takes for each application.

$$Cost^{time}_{marginal}(A) = \sum_{i=1}^{\eta(A)} [\alpha(Ebg(A), \mathcal{Q}(i))\hat{\alpha}(Ebg(A), \mathcal{Q}(i))Cost^{time}_{apply-rule}(Ebg(A), \mathcal{Q}(i))] \tag{4.18}$$

The time it takes to apply a rule to a query instance depends on many complicated factors. One such factor is the antecedents of the rule. Application time increases with the number of antecedents in the rule. If the rule has no expensive variables, this relation is simple; in the other case, however, application time depends on the number of expensive variables, the size of the domain of each variable, the ordering of the antecedents, and other quantities. These factors are complicated enough, and interact enough, that it is impossible to determine the exact function representing the application time of a learned rule. Therefore, for the purposes of this utility analysis, this cost is represented by an unknown function:

$$Cost^{time}_{apply-rule}(R, A) = \mathcal{F}(A, Ants(R), ExpVars(R), |domains(R, ExpVars(R))|, ...) \tag{4.19}$$

EBG Utility Unknowns

The EBG ideal utility calculations involve the following unknown quantities:

- $\mathcal{L}(Ebg(A))$
- $\mathcal{F}(...)$

- $\sum_{i=1}^{\eta(A)} [RSucc(EBG(A), \mathcal{Q}(i)) \times \tau(i)]$
- $\sum_{i=1}^{\eta(A)} \alpha(Ebg(A), \mathcal{Q}(i))$
- $\hat{\alpha}(Ebg(A), \mathcal{Q}(i))$, for $1 \leq i \leq \eta(A)$

The value first of the quantity is assumed to have a negligible cost, since it is multiplied by K_{cons} in the analysis, and K_{cons} is taken to equal 0 as in the caching analysis. The second item is unknown due to the complexity of the function this item represents. The rest of the items require knowledge concerning future events: how successful the rule will be, the time cost of future query instances, and the cases for which the learned rule will be applied.

EBG Utility Estimates

ISM estimates the unknown EBG utility quantities as follows:

$$\hat{\alpha}(Ebg(A), \mathcal{Q}(i)) \approx 1, 1 \leq i \leq \eta(A) \tag{4.20}$$

ISM assumes that if a rule is applicable to a query instance, it will be applied. This is true when:

- the value of the query instance consists of a set of items (instead of a single item), or
- there are no other rules applicable to the query instance. This is true when all the learned rules apply to distinct slot values. Hence, this approximation tends to be reasonable when the learned rules are general enough (relative to the query distribution) that multiple rules are not needed for a single slot.

$$\sum_{i=1}^{\eta(A)} \alpha(Ebg(A), \mathcal{Q}(i)) \approx \frac{|queries(slot(A))|}{|queries(A)|} \tag{4.21}$$

This formula states that the number of times that a rule is applicable over query instance A's *utility calculation timeframe* is approximately equal to $\frac{|queries(slot(A))|}{|queries(A)|}$. Since a rule is applicable whenever its slot equals the slot of a query instance, this approximation is quite accurate.

$$\sum_{i=1}^{\eta(A)} [RSucc(EBG(A), \mathcal{Q}(i)) \times \tau(i)] \approx Cost_{inf}(A) \frac{|ExplanationStructureMatches(A)|}{|queries(A)|} \tag{4.22}$$

From this equation, we see that the benefit derived from the rule $EBG(A)$ is approximated by query instance A's inference time multiplied by a measure of the number of times $EBG(A)$ will be successful before A is requeried – before the utility calculation timeframe expires. This measure is taken to be $\frac{|ExplanationStructureMatches(A)|}{|queries(A)|}$. The numerator of this term is the number of times A's explanation structure has matched the explanation structure of any query instance. Hence, this term gives the average number of times $EBG(A)$ is useful for every query instance A.

Unfortunately, true explanation structures cannot always be generated. Some of Theo's inference mechanisms cannot be represented by a domain theory. If a knowledge base designer implements a novel *method* or special-purpose *toget*, ISM cannot represent these inference mechanisms in an explanation structure – no domain theory represents these mechanisms. To handle this problem, explanation structures are augmented so that their nodes can include *methods* and *togets* in addition to domain theory rules. Augmented explanation structures are called *inference structures*. The inference structure of a query instance A is $InfStruct(A)$. ISM uses inference structures to approximate explanation structures. Hence,

$$\sum_{i=1}^{\eta(A)} [RSucc(EBG(A), \mathcal{Q}(i)) \times \tau(i)] \approx Cost_{inf}(A) \frac{|InfStructMatches(A)|}{|queries(A)|} \tag{4.23}$$

$$\mathcal{F}(...) \approx K_{apply-rule} \times Cost_{inf}(A) \tag{4.24}$$

ISM assumes that the application cost of $EBG(A)$ is proportional to the time needed to infer A. This approximation is very rough. Essentially, ISM assumes that the time difference between applying a learned rule and actually inferring the query instance derives from meta-level slot inference. ISM further assumes that meta-level slot inference takes a fixed proportion of time for the inference. Although rough, this approximation is very inexpensive to calculate, which, given the complexity of F, is very important. If not quantitatively accurate, this estimate is at least qualitatively reasonable.

Rule Generation and Rule Elimination

From the above analysis, ISM takes EBG utility to equal:

$$\begin{aligned} U_{EBG}(A) &= \frac{Cost_{inf}(A)}{|queries(A)|}[|InfStructMatches(A)| \\ &\quad -K_{apply-rule}|queries(slot(A))|] \end{aligned} \tag{4.25}$$

Thus, EBG utility is positive if

$$|InfStructMatches(A)| > K_{apply-rule}|queries(slot(A))| \tag{4.26}$$

ISM uses this formula to determine when EBG should be used to generate rules. Note that the overhead of creating the rule is not included in this formula. Because ISM cannot know how often a rule will be used during the life-time of the system, it considers only marginal costs in this utility calculation. However, to ensure that a rule has a reasonable likelihood of being useful, a threshold criterion must be satisfied before EBG can be applied to query instance A:

$$|InfStructMatches(A)| > 4 \tag{4.27}$$

Because the EBG utility formula considers only the marginal costs of EBG, is can also be used to *eliminate* existing rules. I.e.,

$$U_{EBG}(A) = -U_{unEBG}(A) \tag{4.28}$$

If, after a rule is generated, ISM finds the utility of the rule to be negative – i.e., the overhead of applying the rule outweighs the speedup gained from the rule – ISM can prevent Theo from using it. Hence, ISM uses utility estimates to invoke and revoke speedup mechanisms.

As for caching, ISM calculates EBG utilities for a query instance at the conclusion of the instance's inference. This allows ISM to use the sensing data gathered during in inference. Because of this, even at startup, ISM has data from which to calculate utilities.

EBG Sensors

The following data is used by ISM for its EBG utility calculations:

- $Cost_{inf}(A)$: this sensor is also used for determining the utility of caching, as has been discussed previously.

- $|InfStructMatches(A)|$: ISM generates inference structures on demand for a query instance by constructing a tree of prolog rules, methods, or togets used to infer the instance. The nodes in the tree are determined by following the *expl* pointers for the instance. One the tree is constructed, it is compared with other existing trees to find matches.

- $|queries(A)|$: ISM keeps a counter for each query instance representing the number of times that it has been queried. This data is stored in a subslot of the instance.

- $|queries(slot(A))|$: ISM also keeps a counter for each slot S representing the number of times instances whose slot equals S have been queried. This data is stored in a subslot of S.

The last three sensors are all very long-term; they keep running counts of information over the life of the system. Because of EBG's large initial-cost overhead, it is important to try to guarantee EBG's usefulness if it is invoked in order to amortize this overhead (since it is ignored in ISM's utility estimate). Since this is impossible, keeping a large history of information relevant to EBG's utility calculation gives ISM the best chance of predicting EBG utility correctly. Also, monitoring running totals is more efficient than monitoring over smaller time periods – maintaining a history of the quantity being measured is not needed.

EBG Effectors

ISM invokes EBG on a query instance by setting the value of that instance's *whentoEBG* slot. At the conclusion of the inference, Theo checks the value of this slot. If it has been set, EBG is used to generate a rule.

ISM eliminates a learned rule by reindexing it. After reindexing, Theo's inference methods cannot find the rule and it cannot be applied. Rule reindexing has two advantages:

- If it later appears to ISM that the rule is useful, ISM does not have to regenerate the rule – it can be moved to its original location.

- The rule can be used by BEBG as a template to generate bounded-cost rules.

4.1.3 Bounded-Cost EBG

Factors Influencing BEBG Utility

As described in Chapter 2, BEBG is a variant of EBG. Specifically, BEBG instantiates the expensive variables of rules generated by EBG. This reduces the generality of the rule, but increases its application efficiency. In terms of the generality-efficiency tradeoff, BEBG rules lie between EBG and caching in terms of both generality and efficiency.

Because BEBG is very similar to EBG, the factors that influence EBG utility tend to also BEBG utility. However, there are differences. Since BEBG rules have no expensive variables, the application cost of such a rule is independent of the domain size of any relation. Also, the potential specificity of a BEBG rule means that such a rule may only be useful under specialized query distributions.

BEBG Ideal Utility

Let:

$U_{BEBG}(A)$ = the utility of BEBGing query instance A.
$Bebg(A)$ = the rule resulting from applying BEBG to A.
$|\mathcal{S}(R)|$ = the number of times rule R has been or could have been applied successfully.

BEBG utility is very similar to EBG utility. The following equations are identical.

$$U_{BEBG}(A) = Benefit_{BEBG}(A) - Cost_{BEBG}(A) \tag{4.29}$$

$$Benefit_{BEBG}(A) = \sum_{i=1}^{\eta(A)} [RSucc(Bebg(A), \mathcal{Q}(i)) \times \tau(i)] \tag{4.30}$$

$$Cost_{BEBG}(A) = Cost^{time}_{initial}(A) + Cost^{time}_{marginal}(A) + Cost^{space}(A) \tag{4.31}$$

$$Cost^{space}(A) = K_{cons} \times \mathcal{L}(Bebg(A)) \times \eta(A) \tag{4.32}$$

$$Cost^{time}_{initial}(A) = K_{put} + K_{EBG} \times \mathcal{L}(\mathcal{E}^*(A)) \tag{4.33}$$

$$Cost^{time}_{marginal}(A) = \sum_{i=1}^{\eta(A)} [\alpha(Ebg(A), \mathcal{Q}(i))\hat{\alpha}(Ebg(A), \mathcal{Q}(i))Cost^{time}_{apply-rule}(Ebg(A), \mathcal{Q}(i))] \tag{4.34}$$

The application cost of BEBG rules is independent of the domain size of any relation. Hence,

$$Cost^{time}_{apply-rule}(R, A) = \mathcal{F}(A, Ants(R)) \tag{4.35}$$

BEBG Utility Unknowns

The BEBG unknowns are identical to the EBG unknowns. They are:

- $\sum_{i=1}^{\eta(A)} [RSucc(Bebg(A), \mathcal{Q}(i)) \times \tau(i)]$
- $\sum_{i=1}^{\eta(A)} \alpha(Bebg(A), \mathcal{Q}(i))$
- $\hat{\alpha}(Bebg(A), \mathcal{Q}(i)), 1 \leq i \leq \eta(A)$
- $\mathcal{F}(...)$

BEBG Utility Estimates

As for EBG, ISM makes the following BEBG utility approximations for:

$$\hat{\alpha}(Bebg(A), \mathcal{Q}(i)) \approx 1, 1 \leq i \leq \eta(A) \tag{4.36}$$

$$\sum_{i=1}^{\eta(A)} \alpha(Bebg(A), \mathcal{Q}(i)) \approx \frac{|queries(slot(A))|}{|queries(A)|} \tag{4.37}$$

The following BEBG approximations differ from those of EBG:

$$\mathcal{F}(A, R) \approx K_{apply-rule}|Ants(R)| \tag{4.38}$$

Theo applies rules by calling Theo on each rule antecedent. Since each antecedent has no expensive variables, each Theo call takes only constant time. Hence, the rule application time is taken to be proportional to the number of antecedents in the rule. This approximation is not always accurate; depending on Theo's state, the rule can fail before all the antecedents have resulted in Theo calls. However, the application cost of BEBG rules is small enough relative to that of EBG rules that this inaccuracy, in practice, does not hurt ISM's management policies.

$$\sum_{i=1}^{\eta(A)} [RSucc(Bebg(A), \mathcal{Q}(i))\tau(i)] \approx Cost_{inf}(A)\frac{|InfStructMatches(A)|}{|queries(A)|}\frac{|\mathcal{S}(Bebg(A))|}{|\mathcal{S}(Ebg(A))|} \tag{4.39}$$

The benefit estimate of invoking BEBG is similar to that of EBG. In fact, it is taken to equal the benefit of invoking EBG scaled by factor representing the success rate of the BEBG rule relative to the EBG rule success rate. If we assume that the formula giving the EBG rule benefit is correct, the BEBG approximation is very accurate. It has the drawback, however, of giving the average expected benefit over the entire time ISM has been running. Hence, this measure may be slow to respond to environment dynamics.

It is important to note that ISM cannot initially estimate $\frac{|\mathcal{S}(Bebg(A))|}{|\mathcal{S}(Ebg(A))|}$ with any accuracy.

Bounded-Cost Rule Generation and Elimination

The BEBG utility analysis is used by ISM to generate BEBG rules. However, unlike the EBG situation, ISM does not eliminate BEBG rules. The overhead associated with their use is considered small enough that they will never substantially decrease Theo's performance. This strategy reduces ISM overhead by obviating the need for ISM to monitor BEBG rule use data.

BEBG Sensors

Most of the BEBG utility data monitoring demands are handled by the EBG sensors, which have been discussed previously. However, ISM requires some data not monitored by sensors already discussed. This data is:

- $\frac{|\mathcal{S}(Bebg(A))|}{|\mathcal{S}(Ebg(A))|}$
- $|Ants(R)|$

ISM cannot monitor, or even estimate, the first item with any accuracy until after an EBG rule is generated. After this point, as the rule is being used by the system, ISM can sense for this data by monitoring the rule's usage patterns. This effects ISM's behavior in an important way: *ISM will only apply BEBG to a query instance after EBG has been applied to the instance.* ISM's data requirements for estimating BEBG utility essentially makes EBG a precondition to BEBG.

After EBG generates a rule R, R's expensive variables, if any, are recorded in the knowledge base. This is determined from the structure of R. Then, whenever R is applied successfully, sensors record:

- the number of times R has been successfully applied, $Successes(R)$.

- a history of the instantiation-values of $ExpVars(R)$ during R's successful applications.

- the number of matches between the current instantiation values of R's expensive variables and those recorded in the $ExpVars(R)$ instantiation-value history, $|ExpVarMatches(R, A)|$. This gives the number of times that a bounded-cost version of rule R, whose expensive variables were instantiated with the values needed to successfully solve A, would have been successful since R was generated.

$\frac{|\mathcal{S}(Bebg(A))|}{|\mathcal{S}(Ebg(A))|}$ is taken to equal $\frac{|ExpVarMatches[Ebg(A),A]|}{Successes[Ebg(A)]}$. These two quantities differ only slightly: the latter measures BEBG's relative success rate over the lifetime of the EBG rule; the former measures it over the lifetime of the system.

These sensors are long-term due to BEBG's initial application cost. A larger pool of data gives ISM a better chance of predicting BEBG utility correctly. Long-term sensors are also more efficient than those operating over a shorter term.

Since BEBG cannot be invoked until after EBG is invoked, sensing $|Ants(R)|$ is straightforward: this quantity is equal to the number of antecedents of the rule with at least one non-expensive variable.

BEBG Effectors

As with caching and EBG, BEBG is invoked by ISM via a flag. As the inference for the current query instance is concluding, ISM calculates the utility of applying BEBG to the instance. If ISM chooses to apply BEBG, a flag is set. After the inference ends, Theo checks the value of this flag, and if it is set, BEBG is applied.

4.2 Managing Speedup Mechanisms

As described in the previous section, the following speedup mechanism management actions are available (via effectors):

- *cache*: cache a query instance result

- $\overline{cache}$: do not cache or uncache a query instance result

- EBG: generate an EBG rule on a query instance

- $\overline{EBG}$: eliminate an EBG rule
- $BEBG$: generate a BEBG rule on a query instance

Hence, for each query instance, ISM faces the issue: which action or set of actions should be taken to increase Theo's efficiency? ISM needs speedup mechanism utility estimates for this decision, but exactly how is this decision made?

4.2.1 Possible Management Strategies

An Aggressive Strategy

At first glance, a reasonable management strategy would be to apply every action whose associated utility was positive. However, there are potential problems with this approach. ISM's utility estimates are based on the speedup expectations of each mechanism *in isolation* – the estimates do not take into account the effects between speedup mechanisms if invoked simultaneously. This is a problem in the situations where multiple mechanisms do not give any additional speedup leverage over a single mechanism, or where multiple mechanisms interact negatively. Under these cases, this management strategy is inefficient if the overhead of applying the multiple mechanisms exceeds the overhead of applying a single mechanism.

For instance, if an address A is repeatedly requeried, and the knowledge base is stable, both caching and EBG will have positive utilities. However, caching is sufficient – applying both EBG and caching will not result in more speedup than applying only caching. In fact, using both EBG and caching is less efficient than using only caching in this situation due to the non-trivial initial cost of EBG. This will be referred to as the *initial-cost* problem.

There is another similar but more subtle problem. Suppose a query sequence consisting of two subsequences, S_1 and S_2, occurs repeatedly. S_1 is a sequence of query instances for which EBG has negative utility. S_2 consists of a number of repeated queries to address A under a stable knowledge base. EBG's utility for S_2 is positive, perhaps positive enough to outweigh EBG's negative utility for S_1, in which case a rule would be generated. Caching is also useful for S_2, so ISM would apply caching to A. Unfortunately, caching effectively nullifies EBG's utility for S_2. In this case, EBG will actually decrease architecture efficiency, since the learned rule utility for S_1 is negative. Essentially, EBG is never able to realize its speedup expectations because of caching. However, EBG's potential to slow down the system does materialize. This will be referred to as the *negative-utility* problem.

Both these problems stem from the fact that ISM's utility estimates do not take the effect of other speedup mechanisms directly into account. Hence one mechanism may cause another to actually hurt architecture performance.

A Conservative Strategy

The initial-cost and negative-utility problems show that applying all speedup mechanism management actions whose associated utilities are positive can be a poor management technique. One way of handling this problem is to allow only one management action per query instance. This prevents the unforeseen speedup mechanism interaction problem, since the effects of every management action become visible (through sensing data) before another potentially interfering management action is considered.

Unfortunately, this strategy seems overly restrictive. It can be useful to have the ability to invoke multiple management actions at the same time, since this gives the manager more flexibility. This kind of flexibility allows the manager to modify the architecture's operation at a faster rate, which can lead to better performance during periods when the architecture's environment is changing.

4.2.2 ISM's Management Strategy

ISM's management strategy provides the flexibility of allowing multiple simultaneous management actions while side-stepping the initial-cost and negative-utility problems. These problems arise because the invocation of one mechanism can reduce the utility of another (though this property is not reflected in ISM's utility estimates). If this is the case, these mechanisms are said to *interfere*. In a nutshell, ISM's strategy allows multiple management actions except when they interfere with each other.

To find these interferences, ISM considers the utility function associated with each management action. If the sensing data needed to determine the utility for an action (say, A_1) can be affected by another action A_2, A_1 and A_2 *interact*. If, however, A_2 affects the sensor data of A_1 in a way that can *decrease* A_1's utility, the mechanisms *interfere*. Similarly, if A_1 affects sensor data in a way that decreases A_2's utility, A_1 interferes with A_2.

For example, let A_1 = EBG and $A_2 = \overline{cache}$. Uncaching the query instance can change $Cost_{inf}$, which is used to estimate the utility for A_1. Hence, A_1 and A_2 interact. However, A_2 can only increase $Cost_{inf}$. Increas-

cache $\longleftrightarrow$ EBG
cache $\longleftrightarrow$ $\overline{EBG}$
cache $\longleftrightarrow$ BEBG
$\overline{EBG}$ $\longleftrightarrow$ BEBG

Figure 4.1: Interfering Speedup Mechanism Management Actions

$\overline{cache}$ $\longleftrightarrow$ EBG
$\overline{cache}$ $\longleftrightarrow$ $\overline{EBG}$
$\overline{cache}$ $\longleftrightarrow$ BEBG

Figure 4.2: Non-Interfering Speedup Mechanism Management Actions

ing $Cost_{inf}$ cannot decrease the utility of A_1. Consequently, A_1 and A_2 do not interfere.

On the other hand, let A_1 = cache and A_2 = EBG. Again, there is an interaction between A_1 and A_2. In this case, EBG can decrease the value of $Cost_{inf}$. This can reduce the utility of A_1, which means EBG interferes with caching. Hence, ISM will never invoke EBG and caching simultaneously.

To determine the possible interferences between the management actions, all pairs of actions along with their associated utility estimates have been analyzed similarly. Figure 4.1 shows the interfering actions. All the interferences are due to the $Cost_{inf}$ data.

The non-interfering actions are shown in Figure 4.2. These have been determined from the interfering actions as well as constraints between the complementary actions (i.e., an action and its complement cannot be invoked simultaneously) , and the observation that BEBG can be applied only after EBG has been applied (as discussed in the BEBG utility analysis).

The ISM Algorithm

ISM determines which management action to take according to the following algorithm:

```
At the conclusion of each query instance I:
  For each management action A:
    Estimate the utility of applying A on I.
  Construct a list L of actions sorted from highest to lowest utility
  For each A' in L with associated utility U'
    If U' > 0 and
       A' does not interfere with any other action applied to I
      Apply A' to I.
```

This algorithm combines the flexibility of allowing multiple simultaneous management actions with the safety of allowing only one management action per query instance. ISM's choice of applying the action with the greatest utility shows that ISM utilizes a modified hill-climbing algorithm.

Note that estimating the utility of every management action for every query instance can be prohibitively expensive. Calculating utilities for every query instance implies that ISM's monitors must be applied to every instance, as well. ISM manages this overhead by limiting the query instances for which utilities are calculated and from which data is sensed. These strategies are discussed in the "Managing ISM Overhead" chapter.

4.3 Experiments

In this section, two questions are studied experimentally:

- How accurate are ISM's utility approximations?
- How efficient is ISM's management strategy relative to the aggressive and conservative extremes?

4.3.1 Utility Approximations

Caching Utility Estimate

In Figure 4.3, ideal caching utility is shown against ISM's caching utility estimate. In this experiment, the same top-level query instance Q is repeated 15 times. The knowledge base is stable up to the fourth top-level query, which is Region 1. From the fifth to the ninth top-level query, Region 2, the knowledge base is unstable. Following that, in Region 3, it becomes stable again.

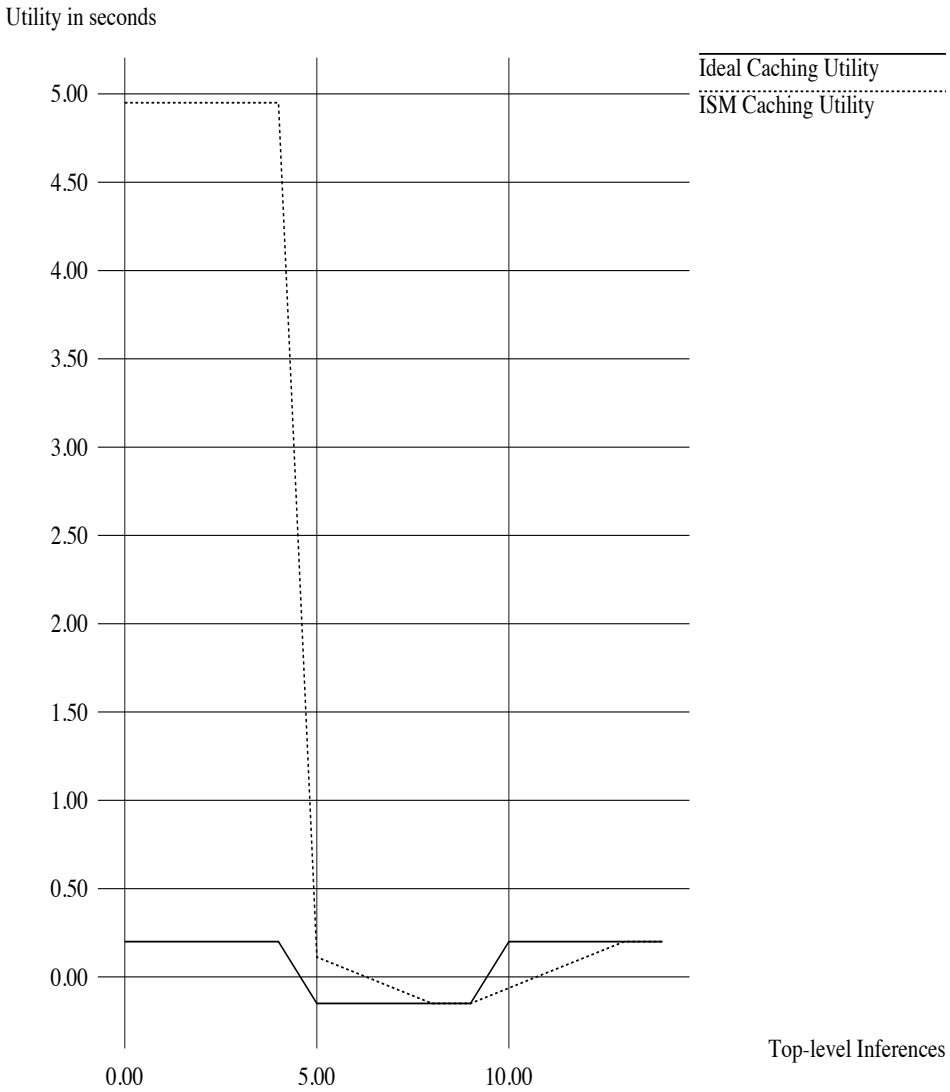

Figure 4.3: Ideal and Estimated Caching Utilities

Consider the figure's ideal caching utility. The initial query requires almost 5 seconds to infer. However, because this initial inference causes various subinferences to be cached, Q's reinference time is only about .2 seconds. Hence ideal caching utility equals about .2 seconds in Region 1. In Region 2, the knowledge base is unstable. Caching Q would result in some TMS overhead, so ideal caching utility is negative. When the KB becomes stable again, in Region 3, caching utility resumes its earlier value.

Now consider ISM's caching utility estimate. ISM's sensors initially find that inferring Q takes almost 5 seconds. Because Q is cached at the conclusion of the first inference, ISM has no opportunity to update Q's inference time – ISM's sensors cannot know a reinference would take far less time. Hence in Region 1, ISM consistently overestimates the utility of caching Q.

In Region 2, at query 5, ISM's estimate falls drastically. During this query, ISM must reinfer Q for the first time. This causes ISM to substantially lower its reinference time estimate, which accounts for almost all of the utility estimate drop. The remainder of the estimate drop is due to the fact that $Prob_{stable}(Q)$ has fallen slightly, since the value of Q is unstable in Region 2.

In Region 2 after query 5, ISM's utility estimate drops until it matches the ideal caching utility. $Prob_{stable}(Q)$ is responsible for this behavior. KB dynamics cause $Prob_{stable}(Q)$ to lower incrementally until it reaches 0 at query 8, whereupon estimated utility equals ideal utility.

This process is reversed in Region 3. $Prob_{stable}(Q)$ increases until it reaches 1.0, causing ISM's estimated utility to increase, until it equals ideal utility.

This figure shows typical utility estimate features. Initially caching utility is considerably inaccurate because ISM's sensors need time to learn Theo's steady-state behavior. ISM's utility estimates also always experience a delayed-reaction during architecture dynamics – from Region 1 to Region 2, for example. This results from the sensor accuracy/reactivity tradeoff points chosen in ISM's design.

EBG and BEBG

In Figures 4.4 and 4.5, ideal marginal utilities are compared with ISM's utility estimates. These experiments have four regions. Region 1 (queries 0 to 4) consists of queries for which a learned rule R is useful. In Region 2 (queries 5 to 9), R cannot be applied. In Region 3 (queries 10 to 14), R can be applied, but is not successful. In Region 4 (queries 15 to 19) R is again

successful.

The ideal EBG utility analysis is straightforward. In Region 1, the rule saves .175 seconds per query. In Region 2, the rule cannot be applied, and hence has no utility. In Region 3, the overhead of applying R unsuccessfully is .025 seconds. Finally, Region 4 gives the utility of the rule when it is successful.

Because EBG and BEBG sensors are long-term, ISM utility estimates for these speedup mechanisms depend on the history of the system. Because many sensor values are averaged over the life of the system, a short history causes utility estimates to be more "jumpy;" a long history causes ISM to react much more slowly to architecture dynamics. The different EBG and BEBG curves reflect different histories. This effect is evident in Figure 4.4. The widely-varying curves result from a shorter sensor history. Because the BEBG and EBG utility estimate curves essentially share common characteristics, only a single explanation for all of the curves will be given.

Assume R is generated from query instance Q. Because R is successful for queries in Region 1, $|InfStructMatches(Q)|$ increases, causing estimated utility to increase. In Region 2, $|InfStructMatches(Q)| = 0$, and the slots of the queries in the region do not equal $slot(Q)$. Hence utility equals 0. In Region 3, $|queries(slot(Q))|$ increases, but because R cannot be used, $|InfStructMatches(Q)|$ stays constant, causing utility to fall. Finally, in Region 4, increasing inference structure matches cause estimated utility to rise.

Note that, initial values of ISM's estimates (for each region) can be very incorrect. This characteristic results because ISM cannot know the future; it must assume that past events essentially guide future events. Because this is an inherent limitation, there is no way to minimize this problem.

When comparing ideal EBG/BEBG utility with estimated utility, we see that in the different regions, ISM's estimates approach the ideal utility calculations. However, because of ISM's long-term sensors, ISM's estimates can never equal the ideal utility calculations – in contrast to ISM's caching utility estimates.

It is evident that ISM's caching estimates are far more accurate than its EBG/BEBG utility estimates. However, estimating the latter quantities is a much harder problem. Furthermore, these estimates are accurate enough that ISM is able to meet its performance goals. ISM's overall performance is demonstrated in the "Experimental Results" chapter.

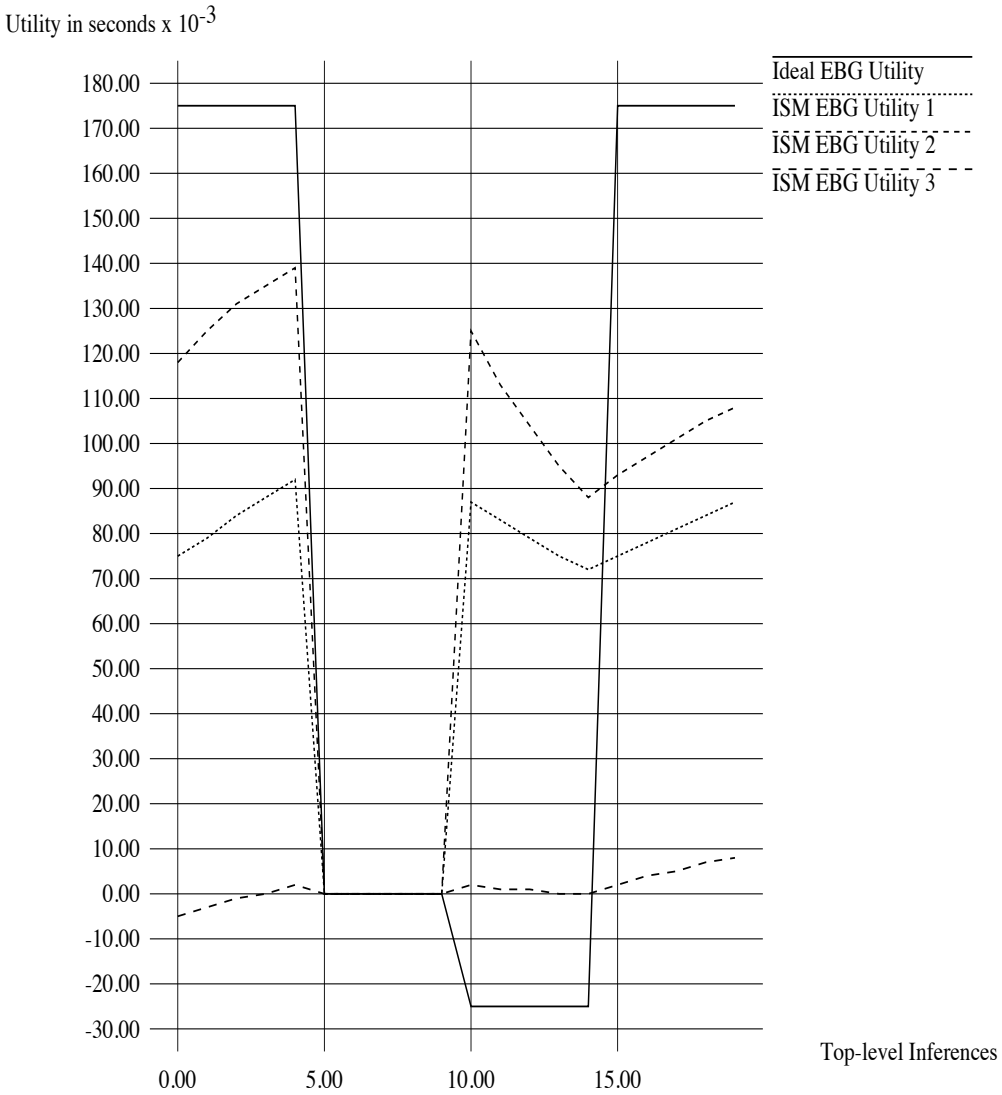

Figure 4.4: Ideal and Estimated Marginal EBG Utilities

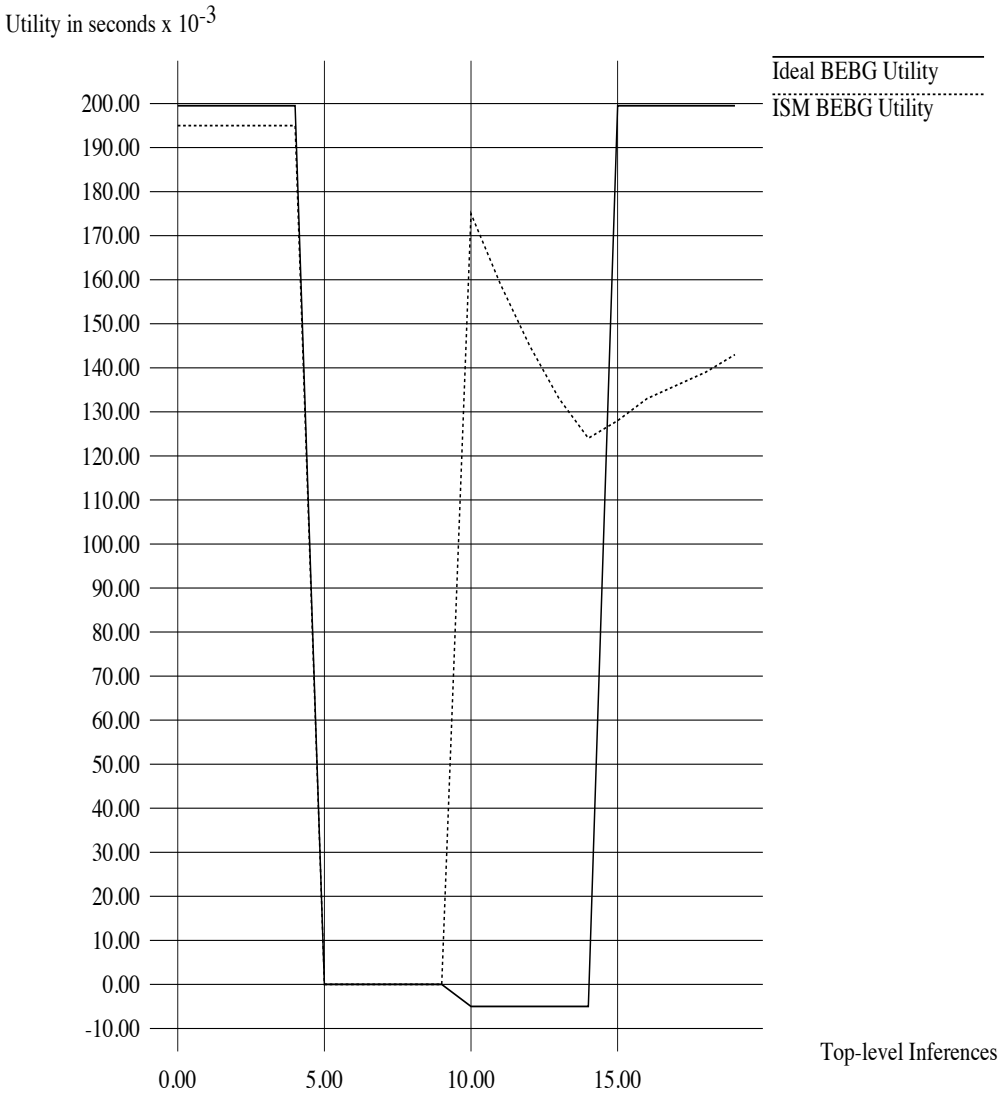

Figure 4.5: Ideal and Estimated Marginal BEBG Utilities

	CAP	MN
Conservative	143.4	47.3
Aggressive	164.0	52.8
ISM	142.0	46.8

Figure 4.6: Management Strategy Performance: inference time (seconds)

4.3.2 Management Strategy

Experiments were conducted using the CAP and MN domains, both of which are described in the "Experimental Results" chapter. These results are summarized in Figure 4.6.

In both domains, ISM's strategy is superior to both the conservative and aggressive strategies. Also, in both domains, the conservative strategy is superior to the aggressive strategy. To some extent, this is to be expected. Functionally, for the speedup mechanisms handled in this thesis, the difference between ISM's strategy and the conservative strategy is small. Rarely, if ever, would ISM need to simultaneously apply $\overline{cache}$ and EBG, $\overline{cache}$ and $\overline{EBG}$, or $\overline{cache}$ and $BEBG$. With a different set of speedup mechanisms, such an ability could be much more useful. However, for caching, EBG, and BEBG, it is not. Hence, the conservative strategy almost equals the performance of ISM's strategy.

On the other hand, it is fairly common for caching, EBG, and BEBG to interfere. Hence, ISM performance is significantly higher than the performance resulting from aggressive management of speedup mechanisms. For instance, in the CAP domain, the aggressive strategy invokes EBG 25 times; ISM's strategy invokes EBG only 9 times. This result, along with performance comparisons, demonstrate the problems with the aggressive management strategy.

4.4 Managing Additional Speedup Mechanisms

This thesis uses the caching, EBG, and BEBG mechanisms to test the efficacy of ISM. However, for an agent such as ISM to be truly useful, it must be possible to augment the agent to handle additional mechanisms. How can this be done? To add a mechanism M is ISM, the following analysis is required:

- Determine the *ideal utility* of M – its fixed and marginal costs and benefits given perfect information about the operation of the architecture.
- *Estimate* the data needed for the ideal utility calculation in some reasonable way. This gives the utility estimate.
- Design sensors that are able to monitor the data required for the utility estimate.
- Determine the speedup mechanisms with which M can interfere.

This analysis allows ISM to estimate the utility of M for any query, and to use M (with the other mechanisms) effectively.

4.4.1 A Brief Example

Consider inference methods. For every query instance Q, Theo retrieves Q's methods, and tries each in order. The first succeeding method is used to infer Q. If methods are ordered poorly, Theo can spend much of its time applying useless methods. A speedup mechanism can be designed to minimize this problem; sort the inference methods of a query instance by their average cost-benefit ratio.

Ideal Utility

When invoked, this mechanism has the following initial time costs:

- Time needed to retrieve methods data
- Time needed to sort methods
- Time needed to store sorted methods

That is,

$$Cost_{initial}(A) = K_{get}\mathcal{L}(|methods|) + K_{sort}(\mathcal{L}(methods))^2 + K_{put} \quad (4.40)$$

These quantities are simple to monitor.

Once the methods are sorted and stored back into the knowledge-base, a time-cost is incurred every time these methods are used. This time-cost equals the inference time of the query. More precisely, let

$$\mathcal{M}(q,m) = \begin{cases} 1 & \text{if query } q \text{ uses methods list } m \\ 0 & \text{otherwise} \end{cases}$$
$$SM(q) \quad = \text{the sorted methods of query } q$$

Then,

$$Cost_{marginal}(A) = \sum_{i=1}^{\eta(A)} \mathcal{M}(\mathcal{Q}(i), SM(A)) Cost_{inf}^{sorted}(\mathcal{Q}(i)) \tag{4.41}$$

On the other hand, methods sorting saves the inference time that would have occurred with the unsorted methods.

$$Benefit_{marginal}(A) = \sum_{i=1}^{\eta(A)} \mathcal{M}(\mathcal{Q}(i), SM(A)) Cost_{inf}^{unsorted}(\mathcal{Q}(i)) \tag{4.42}$$

Because space costs are minor, they have been ignored.

Estimates and Sensors

Let A be a query instance, and $m = SM(A)$ be the methods list under consideration. The following quantities are unknown:

- $Cost_{marginal}(A)$: This can be computed if ISM monitors the success rate and average inference time of each of the methods in m. ISM simply finds the optimal methods ordering and computes the expected application time of the resulting list. This is multiplied by the number of times m is used over the utility calculation time-period, which can be taken be equal $\frac{|applications(m)|}{|queries(A)|}$.

- $Benefit_{marginal}(A)$: This can be determined by computing the average inference time of any problem instance using methods list m, multiplied by the number of times m is applied over the time-period.

These estimates seem to be good approximations of the ideal quantities needed for the utility calculation. Unfortunately, the first estimate is very expensive to compute – perhaps as expensive as actually invoking the methods-ordering speedup mechanism. How can this problem be handled?

The actual quantity of interest in this utility calculation is the net marginal benefit of methods ordering:

$$Benefit^{net}_{mrgnl}(A) = \sum_{i=1}^{\eta(A)} \mathcal{M}(\mathcal{Q}(i), SM(A)[Cost^{unsrtd}_{inf}(\mathcal{Q}(i)) - Cost^{srtd}_{inf}(\mathcal{Q}(i))] \tag{4.43}$$

A first-order approximation can be taken to equal the *expected failure time* of the first method m_1 of m, $FailureRate(m_1)Cost_{app}(m_1)$ where $Cost_{app}(m_1)$ equals the time needed to apply m_1. Obviously this approximation is very rough. However, note that it supports ISM's goal of never decreasing Theo's efficiency. That is, the approximation is a lower bound.

Interestingly, many of the sensors needed by ISM to manage methods-ordering are used for caching, EBG, and BEBG management. In general, utility calculations for multiple speedup mechanisms require similar sensor data. Hence, adding speedup mechanisms is usually not a difficult task.

Interferences

To ensure that no complications or unwanted interactions result from adding additional speedup mechanisms, it is necessary to determine which speedup mechanisms can interfere. Since methods-ordering can decrease the inference time of a query, methods-ordering interferes with caching, EBG, $\overline{EBG}$ and BEBG.

4.5 Summary

In this chapter, ISM's basic speedup mechanism management strategy was discussed. ISM's operation is based on a utility analysis. Although ISM can only *estimate* speedup mechanism utilities due to sensor limitations, the derivation of ISM's utility estimates from ideal utilities show ISM's approximations to be reasonable. Furthermore, experimental data show that ISM's estimates are effective.

ISM's actual management decision strategy entails an analysis of the interfering and non-interfering management actions available to ISM. This strategy combines the advantages of aggressive and conservative speedup mechanism application strategies, and avoids their disadvantages.

The fine-grained experiments conducted in this chapter give a feel for the accuracy of ISM's utility estimates, as well as the efficacy of ISM's decision strategy relative to the conservative and aggressive extremes. Larger-scale

demonstrations of ISM performance can be found in the Chapter 7, Experimental Results.

Chapter 5

Reducing the System Efficiency/Flexibility Tradeoff

The architecture flexibility/efficiency tradeoff was defined in Chapter 1. Reasons for the existence of this tradeoff were also presented. Briefly, a flexible system allows the user to specify the behavior of the architecture at a finer grain-size. However, supporting this flexibility means the architecture must, at run-time, *infer* its own behavior, reducing system performance. Current architectures allow the user to operate at only a single, fixed flexibility-efficiency tradeoff point.

This chapter presents a technique whereby architectures can operate at different tradeoff points, depending on the needs of the user or domain. Moreover, this technique allows the architecture to determine the proper tradeoff point it autonomously. That is, this chapter shows how Theo's inference can be modified to give the configurability of a flexible, slow system, but give the efficiency of an inflexible, fast system – it allows flexibility if required by the domain, but if not required, it *automatically* increases inference efficiency.

Theo's inference mechanism essentially consists of a search through the knowledge base for relevant data. Often, this search is inefficient – many of the inference paths that are explored fail. This chapter discusses an algorithm that ISM uses to avoid unsuccessful search paths in a way that does not modify Theo's semantics, uniformity, or flexibility. Unlike the techniques discussed in the previous chapter, this algorithm does not opti-

mize Theo's inference via a dynamic, run-time analysis. Instead, it analyzes Theo's knowledge base prior to run-time, statically. This analysis allows the algorithm to intelligently prune Theo's inference search paths, increasing Theo's efficiency. I.e., the static analysis determines an *inference boundary* outside which any inference *must* fail. This boundary allows Theo to avoid some search, increasing efficiency. The boundary for any query instance is determined from the methods of the instance as well as the state of the knowledge-base.

This chapter motivates the ISM algorithm by discussing the particular inefficiency in Theo's inference mechanism that the algorithm addresses, then proceeds by explaining the architecture design choices leading to this inefficiency, and shows why Theo's existing speedup mechanisms are insufficient to handle the problem. The algorithm is then described, and an illustrative example is given.

5.1 Theo's Inference

Consider Theo's inference strategy. Theo infers the value of a problem instance (say, P) via a backward-chaining search through the knowledge base for data relevant to the problem. This search is conducted recursively, depth-first. Typically, it terminates with the first recursive inference that returns a value.

Theo's actual search paths are specified in the knowledge base by the *methods* of the problem instance P. Each of P's methods gives a new problem instance (or set of instances) whose value Theo attempts to infer. Note that, due to Theo's uniformity, determining the value of the system-level (or meta-level) *methods* slot of P is itself a new problem instance.

To illustrate, consider a simple example. Assume that P = (square height) and the state of the knowledge base is as follows:

```
(square *novalue*
  (generalizations (rectangle))
  (height *novalue*
    (methods (inherits default.value))
    (default.value *novalue*
      (methods (drop.context inherits)))))

(height *novalue*
  (default.value 5))
```

The methods of P specify that inheritance and default-value should be applied to infer P. Hence, Theo begins its search by applying inheritance; this results in the new problem instance or subgoal (rectangle height). This inference path may lead to even more subgoals, one of which may be successful, terminating the top-level inference. Assume, however, that the inheritance method is not successful for P – i.e., (rectangle height) = *inferred.novalue*. At this point, Theo tries the default-value method for P, causing Theo to subgoal on (square height default.value). Theo proceeds by applying drop.context, subgoaling on (height default.value) which finally returns a value, terminating the inference. This inference search is depicted in Figure 5.1.

For the purposes of discussing Theo's inference, it is useful to define several quantities. For a top-level query with address A, let

```
  T(A) =  A's inference tree, including meta-level inferences
  I(A) =  set of addresses of immediate inferences
          resulting from query to A
 I*(A) =  transitive closure of I(A), i.e., I(A), I(I(A)), ...
  B(A) =  leaves of T(A) which contain values from which the
          value of A is derived
```

For Figure 5.1,

```
    A  =  (square height)
 T(A) =  entire graph
 I(A) =  {(rectangle height) (square height default.value)
           (square height methods)}
I*(A) =  T(A) - A
B(A)  =  {(height default.value)}
```

5.2 Inefficient Meta-Level Inference

Although this inference mechanism is very natural, it can be quite inefficient. Exploring a search path can require many recursive inferences, even in a very simple knowledge base. If the search path is unsuccessful, the time spent exploring that path is wasted. For example, in a knowledge base similar to that presented above, determining the value of (square height) requires 360 inferences. Of these, Theo requires 202 inferences to explore the unsuccessful

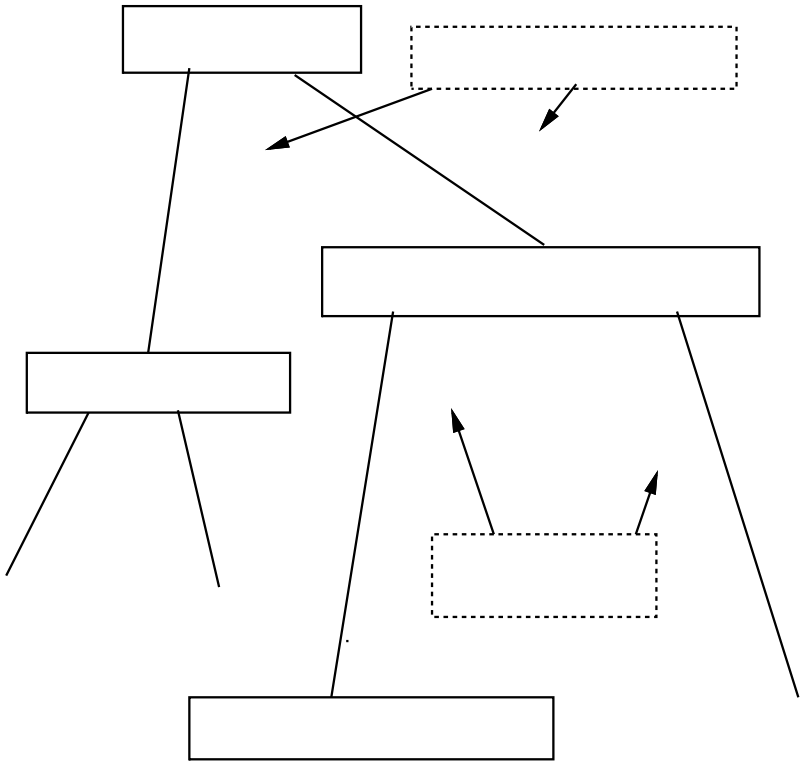

Figure 5.1: Inference Paths for (square height)

inheritance search path. Why so many for such a simple knowledge base? Recall from Chapter 2 that Theo's behavior is largely specified by data in the knowledge base, implying that Theo must determine its behavior via run-time system-level inference. Theo is *so* flexible, in fact, that the vast majority of inferences are system-level inferences. In the above example, 193 of the 202 inferences are meta-level inferences. Typically, this ratio is much lower, but still significant. In the CAP domain, which is described in Chapter 7, system-level inferences outnumber the "ground-level" inferences by a factor of 6.

From this data, it is clear that Theo's efficiency is limited primarily by the amount of meta-level inference. Recall that, due to Theo's uniformity, Theo handles meta-level inference the same way it handles ground-level inference. In particular, Theo uses *methods* to determine its search strategy when inferring system-level slot values. One of the sources of meta-level inference inefficiency results because the methods of meta-level slots have not been optimized. That is, methods often fail because Theo's knowledge bases typically specify methods for meta-level slots at only a very coarse grain-size. Unfortunately, because of the sheer volume of system-level slots and query instances, it is virtually impossible for users or knowledge-base designers to optimize the methods of meta-level query instances.

5.3 Meta-Level Inference and Architecture Flexibility

The vast amount of meta-level inference that Theo must conduct decreases Theo's efficiency considerably. Why have all this meta-level inference? Why was Theo designed in this manner? Although wasteful from the efficiency point of view, Theo's meta-level operation derives from some of Theo's fundamental design principles: flexibility and a highly uniform representation. Theo's representation allows beliefs to be held about any problem, whether the problem is part of the task domain or whether the problem involves Theo's own operation and inference strategies (i.e., the meta-level). This makes Theo very flexible. Since Theo's operation is governed by its beliefs as to how to approach problems, the ability to maintain distinct beliefs for every problem instance implies that Theo's operation can be configured for a particular problem instance independently of any other problem instances. This also gives Theo the ability to reason about itself. Unfortunately, this capability means that meta-level, or system inference is treated similarly

to ground-level, or task domain inference. Consequently, just as ground-level inference results in several system-level inferences, these system-level inferences result in even more system-level inferences. One of Theo's most interesting features is its ability to reason about itself. The price for that ability is the meta-level inference "bottleneck."

To understand the extent of Theo's configurability and flexibility, consider the knowledge-base fragment Figure 5.2. This knowledge base demonstrates fine-grain control over the *whentocache* slot. In this knowledge base, four different caching strategies are used for different intersecting classes of problem instances. Because the primary whentocache inference method is inheritance, the most specific class contains all problem instances that are specializations of (square height whentocache). Those belonging to this class, such as (square1 height whentocache) use the sq-cache caching strategy.

The next larger class contains all specializations of (rectangle height whentocache), such as (rect1 height whentocache). The caching strategy of this class of query instances is rect-cache. Note that members of the (square height whentocache) also belong to this larger class. However, membership in smaller classes have precedence over membership in larger classes. Hence, (square1 height whentocache) uses sq-cache rather than rect-cache.

The secondary whentocache inference method is drop.context. Hence, all problem instances of the form (<entity> height whentocache) are contained in the next class. Again, this class contains the previous two.

The largest, and final class contains all problem instances. The methods of whentocache specify that default-cache is the caching strategy for all query instances not contained in the previous three classes.

This example shows how Theo can be configured to use four different caching strategies in different situations. Theo's methods provide a powerful way in which problem classes may be defined. Although this example deals with the whentocache system slot, most of Theo's other system slots operate in an identical fashion.

The above knowledge base also shows how Theo's meta-level inference can be inefficient. Typically most whentocache query instances will belong to the largest class – the class using the default-cache strategy. However, when inferring values for members in this class, (whentocache methods) specifies that Theo must always first apply the inheritance method. In this situation, inheritance will always fail.

```
(square *novalue*
  (generalizations (rectangle))
  (height *novalue*
    (whentocache sq-cache)))

(square1 *novalue*
  (generalizations (square)))

(rect1 *novalue*
  (generalizations (rectangle)))

(rect2 *novalue*
  (generalizations (rectangle)))

(rectangle *novalue*
  (height *novalue*
    (whentocache rect-cache)))

(height *novalue*
  (whentocache height-cache))

(whentocache *novalue*
  (methods (inherits drop.context default.value))
  (default.value default-cache))
```

Figure 5.2: A Knowledge-Base Fragment

5.4 Using Speedup Mechanisms to Increase Efficiency

We have seen that Theo's design leads to large amounts of meta-level inference, much of which is inefficient due to nonexistent methods optimization. Can any of Theo's speedup mechanisms help in this situation?

5.4.1 Caching

Caching provides a partial solution. Caching allows Theo to remember the results of all of the recursive subqueries initiated by the top-level query. The results of these subqueries can be general; consequently, if cached, they may be reused in the future, saving Theo some re-inference time.

For example, consider the following knowledge base:

```
(box1 *novalue*
  (generalizations (box)))

(box2 *novalue*
  (generalizations (box)))

(whentocache *novalue*
  (methods (inherits drop.context default.value))
  (default.value caching.on))
```

Assume Theo first infers the value of (box1 height whentocache). This value will be caching.on. However, before finding this value, Theo applies the inheritance method, which fails. While applying this method, Theo finds that (box height whentocache) = *inferred.novalue*. If it later must infer the problem instance (box2 height whentocache), Theo must once again infer (box height whentocache). If this value was cached, the time spent searching this inference path would be saved.

Note that caching is only useful as a speedup mechanism if the values cached are stable. Does this pose a problem? The inference bottleneck that this chapter is concerned with deals only with meta-level inference. Fortunately, meta-level values are very stable, allowing caching to effectively increase meta-level inference efficiency.

On the other hand, however, caching does not provide a perfect solution. In domains in which there is not much query instance repetition, the generality of the cached values can be very small relative to the query distribution

– previously cached values may not be reused very frequently. The CAP domain, described in Chapter 7, has this characteristic. ISM's algorithm is useful under all conditions, but especially in this situation.

5.4.2 EBG

EBG is much less effective than caching for increasing meta-level inference efficiency. One approach is to attempt to learn control knowledge. EBG could be used to learn control knowledge in two different ways: by learning the situations under which methods are successful, and by learning situations under which methods are unsuccessful. The semantics of Theo, however, dictate that only the latter strategy is useful. Theo's methods are usually treated as ordered lists, rather than sets. Consequently, to preserve correct Theo behavior, it is necessary to apply the methods in order, unless they are *known* to be unsuccessful.

For example, assume that the following methods are to be applied to a particular problem instance: (inherits drop.context default.value). Assume that an EBG rule posits that default.value is a successful method. Further assume that it is not known whether inherits or drop.context is successful or unsuccessful. In this situation, Theo must first apply inherits even if it knows that default.value will be successful. That knowledge is not useful. On the other hand, if another EBG rule shows that inheritance is unsuccessful, Theo can skip that method.

Unfortunately, using EBG to learn unsuccessful methods results in a very "bushy" rule, since the result is a proof of failure. This kind of rule is very expensive to generate, and tends to be rather specific. Thus, in practice, EBG does not increase Theo's meta-level inference efficiency.

5.5 Algorithm Overview

The previous section shows some of the limitations of using Theo's speedup mechanisms to increase meta-level inference efficiency. ISM uses a different approach. It optimizes Theo's inference by determining, prior to run-time, some of the search paths – i.e., methods – that cannot be successful. This information is utilized at run-time to prune Theo's search, resulting in more efficient inference. Hence, instead of applying speedup mechanisms to make Theo more efficient, ISM modifies Theo more fundamentally by altering Theo's actual search mechanism.

The idea is to determine before run-time, for a problem class (i.e., set of problem instances) P, the *value-containing* addresses in the knowledge base from which the values of elements of P can be computed. This set of addresses is, by definition, B(P). Problem classes for query instances are organized by the slots of those query instances; all problem instances with the same slots belong to the same class. Hence, a problem instance (x y...S) belongs to the class S. Notice that every query instance belongs to one and only one problem class. For the knowledge base in Figure 5.2, let

```
P = whentocache
```

Then,

```
B(P) = {(square height whentocache)
        (rectangle height whentocache)
        (height whentocache)
        (whentocache default.value)}
```

Notice that to infer the value of an element of P, Theo must use the value of at least one of the addresses in B(P). Also, Theo cannot infer the value of an element of P from addresses not in B(P). ISM uses B(P) to determine the methods that can possibly be used to successfully infer values of elements of P – i.e., the methods which lead to inference paths that involve addresses in B(P). These are called P's *potentially successful methods*. Other methods cannot be successful. ISM uses these *failing methods* to prune inference of elements of P by simply not applying any of P's failing methods.

ISM determines the potentially successful methods of P from B(P) by examining the form of the addresses in B(P) and the structure of the knowledge base. For instance, if (box height default.value) is an element of B(P), then default.value is a method that can potentially succeed, since the default.value method results in queries of addresses whose slot equals "default.value". ISM uses this kind of reasoning to determine whether inheritance, drop context, and default value can succeed for each problem instance, and uses this knowledge to prune the methods search path.

In a sense, this algorithm can be considered a form of control-knowledge learning. However, instead of learning at run-time via observation, it discovers control knowledge using a "knowledge-base-preprocessing" strategy.

5.6 Algorithm

Let S be a slot. Let PC(S) be the set of all problem instances whose slot equals S – i.e., all problem instances with addresses of the form (...S). If address A equals (x y ....z), let $right(A)$ equal z and $left(A)$ equal x.

Prior to run-time, ISM:

1. Finds the set of inference methods of the elements of PC(S) by mimicking Theo inference for the problem instance (S methods). Call this set M(S).

2. Determines from M(S) the set of all value-containing addresses in the knowledge base needed to compute the value of any element of PC(S). Call this set B(PC(S)). Elements of B(PC(S)) are said to be *relevant* to PC(S).

   To do this, ISM finds all existing (i.e., "non-virtual") addresses in the knowledge base that have values, and collects the relevant ones according to the following rules:

   - default.value $\in$ M(S) $\Rightarrow$ addresses of the form (<entity> S default.value) are relevant
   - inheritance $\in$ M(S) $\Rightarrow$ addresses of the form (<entity> S) are relevant
   - drop.context $\in$ M(S) $\Rightarrow$ addresses of the form (<entity> S) are relevant

3. Partitions all possible query instances into classes. The classes have the following property: the slots of all members of a class are equal. For each problem class P whose members' slots equal S generate from B(P) a predicate over inference methods M $\in$ M(S) determining if M cannot successfully infer any element of P. This predicate is denoted $F_P(M)$.

   To understand how this is done, consider the elements in B(PC(S)) for some S. If for all b $\in$ B(PC(S)), left(b) has no specializations, then the inherits method will always be unsuccessful for PC(S) – i.e., $F_{PC(S)}(inherits)$ equals true. Similarly, if for all b $\in$ B, right(b) $\neq$ default.value, the default.value method will never be successfully applied, and $F_{PC(S)}(default.value)$ = true.

   ISM uses a more sophisticated form of this idea. In the previous illustration, the problem classes were defined at a coarse grain. The actual ISM algorithm defines smaller problem classes, which can lead to more inference search pruning. The problem classes are defined to have the following properties:

- All members of a class have equal effective address lengths
- The slots of the effective addresses of all members of a class are equal.

where the effective address of a problem instance (x1 x2...xM xN) is defined as (x1 x2...xM) if xN = default.value, and (x1 x2...xM xN) otherwise.

For such a problem class P whose members have slot S and effective address length L, B(P) = B(S) $\bigwedge$ P. The unsuccessful method predicates are defined as follows:

$$U_P(inherits) = \begin{cases} true & \text{if } \forall b \in B(P), left(b) \text{ has no specializations} \\ false & \text{otherwise} \end{cases}$$

$$U_P(default.value) = \begin{cases} false & \text{if } \exists b \in B(P) \text{ such that } right(b) = \text{default.value} \\ true & \text{otherwise} \end{cases}$$

$$U_P(drop.context) = \begin{cases} false & \text{if } \exists \text{ problem class P' whose members' slots equal S and whose members' effective address lengths equal } l, \\ & 0 < l \leq L, \text{ such that } F_{P'}(M) = \text{false for some } M, \text{ where } M \in \\ & \{\text{inherits, default.value, drop.context}\}. \\ true & \text{otherwise} \end{cases}$$

At run-time: Assume Theo is attempting to infer an answer for a problem instance that belongs to problem class P, by applying method M. If $F_P(M)$ = true, M is not applied to the problem instance.

5.7 An Example

Consider the following knowledge base:

```
(height *novalue*
  (generalizations (slot)))

(physobj *novalue*
  (generalizations (root)))

(rectangle *novalue*
  (generalizations (physobj)))

(square *novalue*
  (generalizations (rectangle)))

(whentocache *novalue*
  (generalizations (theoslotslot))
  (default.value caching.on)
  (methods (drop.context inherits default.value)))
```

Assume Theo must infer the value of (square height whentocache). Hence, S = whentocache. Prior to run-time, with S = whentocache, ISM:

1. Determines that M(whentocache) = (whentocache methods) = (drop.context inherits default.value)

2. Finds that B(S) = {(whentocache default.value)}

3. Constructs a frame representing the successful and unsuccessful methods of each problem class associated with the whentocache slot. See Figure 5.3. This frame states that default.value is useful for addresses of length 2, implying that drop.context is useful for whentocache problem classes with address lengths greater than 2. The frame also states that inheritance is not useful for addresses of length 2. In this representation, inheritance and default.value are taken to not be useful for problem classes that are not listed in the frame.

```
(successful-methods *novalue*
  (whentocache *novalue*
    (2 *novalue*
      (default.value? t)
      (inheritance? nil)))
```

Figure 5.3: Useful and Useless Methods for Height

An Inference Trace

Below is an abridged sequence and description of inferences resulting from the (square height whentocache) inference using ISM's optimization algorithm. The first number on each line indicates the Theo inference recursion level. This number is followed by a caret. A forward caret > indicates the initiation of an inference, which is followed by the inference instance address. A backward caret < indicates the conclusion of an inference, which is followed by the inference result. Comments are preceeded by semicolons.

```
;; top-level inference
1> (square height whentocache)
 ;; Theo determines how to infer the top-level query
 2> (square height whentocache methods)
 ;; inference methods are returned
 2< (drop.context inherits default.value)
 ;; Theo tries the first method
 2> (square height whentocache drop.context)
  ;; ISM algorithm is invoked, drop.context is found to be useful
  ;;  for the query and a recursive query instance is generated
  3> (height whentocache)
   ;; As before, methods are inferred
   4> (height whentocache methods)
   4< (drop.context inherits default.value)
   ;; Theo tries the first method on the recursive query
   4> (height whentocache drop.context)
   ;; ISM algorithm is invoked, drop.context found to be ***
   ;;  unuseful for the query and it is terminated
   4< *novalue*
   ;; Theo tries the second method
```

```
    4> (height whentocache inherits)
    ;; ISM algorithm is invoked, inherits found to be ***
    ;;  unuseful for the query and it is terminated
    4< *novalue*
    ;; Theo tries the third method
    4> (height whentocache default.value)
     ;; ISM algorithm is invoked, default.value found to be useful
     ;;  for this query instance -- whose effective address
     ;;  equals (height whentocache).  Recursive query is initiated
     ;;  and Theo finds query's methods
     5> (height whentocache default.value methods)
     ;; Methods are found
     5< (inherits drop.context)
     ;; Theo tries first method
     5> (height whentocache default.value inherits)
     ;; ISM algorithm is invoked, inherits found to be unuseful ***
     5< *novalue*
     ;; Theo tries second method
     5> (height whentocache default.value drop.context)
      ;; ISM algorithm is invoked, drop.context is applied
      6> (whentocache default.value)
      ;; An answer is found and returned to the top level
      6< caching.on
     5< caching.on
    4< caching.on
   3< caching.on
  2< caching.on
 1< caching.on
```

In this trace, pruned search paths are denoted by ***. As the figure shows, ISM is able to avoid three unsuccessful search paths. Because these unsuccessful paths result in many inferences, the time savings resulting from ISM is substantial. Using the ISM algorithm, the above inference takes 16 milliseconds. Without the algorithm, the inference takes 333 milliseconds.

5.8 Algorithm Limitations

This algorithm has several limitations.

- Most seriously, the algorithm must understand the methods being used. Currently, only *inheritance*, *drop-context* and *default-value* are understood. In particular, this algorithm does not handle the *prolog* method, since *prolog* can utilize arbitrary rules. However, since this algorithm is most useful for system slots, and system slots typically only use the inheritance, drop.context and default.value methods, the algorithm is still performs very well.

- As presented, the algorithm assumes that B(P) for all problem classes P is stable. This insures that the pre-run-time knowledge base analysis remains valid during runtime. In practice, this assumption can be relaxed by inserting a monitor that senses changes to B(P). If B(P) is modified, ISM can either:

  1. Stop using the algorithm for this slot or
  2. Recompute the unsuccessful methods for S

- Finally, the algorithm assumes that the methods for all the members of a problem class P (whose slots are S) are equal to the problem instance (S methods). This tends to be true almost all of the time, so this restriction is not too severe.

5.9 Summary

This chapter presented a knowledge-base analysis and inference technique that combines architecture flexibility with inference efficiency. Moreover, this scheme *adapts* to the requirements of the domain. Finally, a simple experiment was conducted, giving some evidence of the effectiveness of this technique.

Chapter 6

Managing ISM Overhead

6.1 The Problem

As described thus far, ISM makes good run-time speedup mechanism management decisions. However, overall system performance might not reflect superior speedup management utilization. ISM's run-time overhead – the time needed to calculate each speedup mechanism's utility and monitor for relevant data – can swamp any speedup gained from intelligent utilization of speedup mechanisms, despite the inexpensive nature of ISM's sensors and utility estimates. Why is this?

The answer to this question lies with Theo's inference mechanism. Theo's operation is such that high-level inferences require many sub-inferences. Each sub-inference is fairly inexpensive and does not actually perform much work. Therefore, Theo's lack of speed derives from the sheer number of sub-inferences needed for a query, rather than any complicated processing performed by Theo during each sub-inference. Recall, however, that ISM's management strategy as described thus far requires ISM to monitor data for, and calculate speedup mechanism utilities for *every* subinference. The plethora of sub-inferences needed by Theo makes ISM overhead substantial.

To be effective, a speedup mechanism manager such as ISM must not only make intelligent management decisions, but must also minimize its own cost. This chapter describes two strategies by which ISM reduces its sensing and utility calculation overhead: adaptive sensing and phasic sensing.

6.2 Adaptive Sensing

One obvious idea for reducing ISM overhead is to monitor and compute utilities for only a subset of Theo's inferences. This is this idea behind ISM's adaptive sensing strategy. An analysis of Theo's inference patterns shows that a speedup mechanism application is only useful for a small percentage of Theo inferences. Hence sensing and utility calculations can be restricted to this set of inferences. Unfortunately, this set is dynamic and changes with Theo's inference patterns. By observing these run-time patterns, ISM is able to determine which query instances to monitor and adapt its monitoring to the environment dynamics. This section describes the assumptions necessary for this sensing strategy, their justifications, and ISM's actual adaptive sensing techniques.

Adaptive Sensing Assumptions

In order to discuss the assumptions needed for ISM's adaptive sensing strategy, it is necessary to define two terms. A query instance is *novel* if it has not been queried in the past. Two query instances are *similar* if their explanation structures overlap – if they have overlapping subproblems. The more these explanation structures overlap, the higher their *similarity level*. If a query instance is similar to a previous query, some portion of the work needed to infer the current instance has been done during the inference of the previous instance.

ISM's adaptive sensing strategy is based on the following three assumptions:

- Consider the entire set of inferences (and sub-inferences) conducted by Theo for a particular domain. If an appropriate caching strategy has been used, only a small percentage of the query instances in this set has been queried multiple times.
- After a small number of top-level queries, almost all novel query instances have a high level of similarity to at least one previous query instance.
- Almost all query instance values in any Theo domain or knowledge-base are stable.

Justifications

ISM's first assumption is based on the observation that of the inferences resulting from a top-level query instance, relatively few of them are ground-level inferences – most of them system-level inferences. Why is this? Theo depends on system-level knowledge base information to guide its operation. A large percentage of system-level query instances are very specific, relevant to only one higher-level inference, reducing the likelihood that such instances will be requeried. Such instances could only be requeried if they were queried by the user, or if the higher-level inference was unstable. However, users virtually never query system-level instances – rarely are users interested in the knowledge base "programming" that specifies Theo's operation. Additionally, system-level values are generally very stable. Users typically do not wish to modify the way the system operates. This further lowers the chances that these kinds of instances would be requeried.

For example, consider the query instance (tom daughters fixedmethods). This instance is useful only for inferring (tom daughters methods). If this latter instance is cached, there is virtually no chance that (tom daughters fixedmethods) will be requeried.

ISM's second assumption also derives from Theo's high level of system-level inference. Theo depends heavily on system-level inference. In fact, in the CAP domain described in the "Experimental Results" chapter, Theo infers about 10 times more system-level slots than user-defined slots. This means that on average, every inference of a user-defined query instance results in 10 system-defined query instances – i.e., the bulk of *any* inference consists of system-level inference. System-level inferences rely heavily on the *inherits* and *drop.context* methods. Many system-level query instances "map" to identical addresses using these two methods. For example, both (tom daughters fixedmethods) and (bob daughters fixedmethods) map to (daughters fixedmethods) using drop.context, and (male daughters fixedmethods) using inherits. These "convergent mappings" between query instances implies a similarity between instances – (daughters fixedmethods) and (male daughters fixedmethods) are overlapping subproblems for both instances. Since generalization hierarchies and contexts are usually shallow, there tends to be a high level level of similarity between novel system-level instances and previous instances. Hence, similarities between current and past query instances result from Theo's large percentage of system-level inference and the convergent inference mappings that are typical for system-level query instances.

As mentioned previously, users typically never modify system address values because there is never a need to modify system behavior. Similarly, Theo itself does not modify system address values. Hence, system-level address values are very stable. Because system-level inference accounts for about 90 percent of Theo inference, at least 90 percent of the values resulting from Theo inference are stable, which justifies the third assumption.

Consequences

The three assumptions allow ISM to limit the number of query instances to consider for speedup mechanism application, reducing the sensing and utility calculations, and hence lowering ISM overhead. Why? Caching can only be useful for repeated query instances. Therefore, the first assumption posits that caching need be considered for only a subset of Theo's inferences, since only a fraction of problem instances will ever be requeried.

The second and third assumptions imply that most novel inferences tend to become less costly over time. According to the second assumption, after the system has run for some time, it becomes difficult to encounter a novel query instance that is not similar to some past query instance. If the subproblems of a novel query instance overlap with past subproblems and the values of the overlapping subproblems are stable and have been cached, these novel inferences have been partially solved. Since the third assumption in fact posits the stability of most query instance values, as a consequence, in general novel inferences become less costly to infer over time.

This conclusion is important with regard to managing BEBG and EBG. It states that the caching speedup mechanism is powerful enough to increase system efficiency for virtually all query instances, because of the *overlapping subproblem* and *stable subproblem value* characteristics of queries. This substantially decreases the potential utility of EBG and BEBG – these speedup mechanisms would be expected to be more useful if these characteristics weren't true. Because caching has a potentially large negative effect on the utility of EBG and BEBG, these mechanisms are presumably most useful in situations under which caching is not a useful speedup mechanism – when query instances values are unstable. Hence, in some sense, the latter two assumptions provide a means by which ISM can be selective in choosing which query instances to consider for EBG or BEBG.

These arguments show that ISM should consider only a subset of query instances when making speedup mechanism management decisions. Theoretically, then, a strategy incorporating these arguments can reduce ISM's

overhead, but only if ISM can determine which query instances are most appropriate for speedup mechanism application.

Overhead Management and Speedup Goals

The argument that EBG and BEBG need be considered for only a subset of Theo's inferences may seem questionable. After all, even if caching is expected to reduce the utility of EBG and BEBG, might these mechanisms be useful regardless? Is it acceptable that ISM considers only some inferences for EBG and BEBG based on a "hand-wavy" rationale?

Overhead management is intimately connected with the goals of ISM, and so this question must be considered in the context of ISM's speedup goals. ISM does not seek to make the optimal speedup mechanism management decisions. It seeks only to increase system performance relative to Theo's default performance. This allows ISM to disregard the situations under which speedup is possible, but unlikely. In some sense, ISM's speedup goals allows it to be conservative in its management strategies.

6.2.1 Strategy

The previous sections have shown that caching, EBG, and BEBG need be considered for application for only some query instances. Thus, ISM now has the task of determining which instances are most appropriate to consider and which to ignore. Because caching, EBG, and BEBG are most useful under different situations, the criteria used to determine when these mechanisms might lead to speedup are different for each mechanism. In this section, these criteria are discussed.

Caching

The adaptive sensing strategy for caching views all query instances that have a reasonable likelihood of being *requeried* as candidates for caching. There are four situations under which there is a reasonable likelihood of this happening for a query instance Q:

- Q is a top-level query. ISM assumes that any top-level query instance can be requeried.
- Q has multiple dependents. This means Q's value is needed for multiple higher-level inferences, so ISM assumes that future inferences may need to infer the value of Q.

- Q has an unstable dependent. ISM assumes that Q's dependent D might be requeried in the future. Because D's value cannot remain cached (since it is unstable), when Theo attempts to infer D, Q will be requeried.

- An expensive rule is being applied, causing Q to be inferred as a subproblem.

ISM calculates caching utilities and monitors data needed to calculate these utilities for the query instances that satisfy any of these conditions. This strategy is adaptive because the set of query instances for which ISM must calculate utilities continually shifts. This means that determining these conditions requires more sensors to be embedded into Theo. Unlike the previously discussed sensors, these determine when ISM should be invoked, and are called I-sensors. The monitors that collect data for ISM's utility analyses that were discussed previously will be called D-sensors.

Note that ISM must invoke its caching I-sensors on *every* problem instance to determine the instances on which the D-sensors should be used. Hence, the adaptive sensing strategy *trades off* expensive D-sensing costs with relatively inexpensive I-sensing costs. With this sensing strategy, ISM avoids invoking the D-sensors. Unfortunately, because the D-sensing must be *adaptive*, an additional "layer" of sensors – the I-sensors – must be added to the system.

I-Sensors for Caching

ISM's I-sensors for caching are implemented as follows:

- Top-level query sensor: this determines whether a query instance is a top-level query. At the beginning of every inference, the Theo stack, which keeps track of all of Theo's recursive calls, is checked. If the stack is empty, the inference is a top-level query and its address is stored in a global variable.

- Multiple dependents sensor: this senses whether a problem instance Q has multiple distinct dependents. This is monitored approximately and indirectly. A fixed-size queue containing the last 100 query instances is maintained by ISM. To check Q for multiple dependents, ISM searches the queue for instances of Q, and for each instance, returns its "predecessors" in the queue. Usually this predecessor repre-

sents an inference resulting in Q's query. If there are multiple distinct "predecessors," Q is assumed to have multiple dependents.

- Unstable address sensor: this senses if a query instance has ever been unstable in the past. ISM saves the value of every novel query instance in the knowledge base. If the query instance is repeated in the future, the answer is compared to its original value. If it has changed, this instance is tagged as unstable.

- Unstable dependent sensor: this determines if a dependent of a query instance Q is unstable. ISM finds Q's dependent by looking at the Theo stack, then checks if it is unstable using the unstable address sensor.

- Expensive rule application sensor: rules are applied by Theo's prolog rule interpreter. Expensive rules can be determined by their syntactic form. Whenever the interpreter attempts to apply an expensive rule, a flag is set.

ISM's multiple dependents sensor is approximate – it is not always accurate. Note that using the *dependents* system slot to monitor for this information does not work; the *dependents* slot of a query instance Q saves information only if Q is cached. Also, ISM's sensing scheme is more space-efficient than monitoring dependents directly. Furthermore, the sensor's failure modes are not very harmful. If it errs by returning spurious dependents, this will result in a loss of efficiency, but a negligible one as long as it does not err frequently. Although the fixed-size queue can make this sensor disregard some dependents, ISM assumes that only recent inference patterns are significant, which allows ISM's behavior to shift with architecture environment dynamics.

EBG

As discussed earlier, due to the high D-sensing costs associated with EBG as well as ISM's speedup goals, ISM considers using EBG on only a subset of the problem instances for which the mechanism could be useful. However, the subset that is considered contains the problem instances for which the mechanism is the most useful.

ISM considers EBG potentially useful for a problem instance Q if:

- Q is unstable *and*

- *Either* Q is a top-level query
- *Or* $|queries(slot(Q))| >$ threshold

As for caching, if a problem instance satisfies the above criteria, ISM's EBG D-sensors are applied.

ISM's threshold criterion allows ISM time to make caching decisions, and time for these decisions to take effect before EBG is considered.

EBG I-Sensors

ISM requires the following I-sensors for EBG:

- Unstable address sensor: described earlier.
- Top-Level query sensor: described earlier.
- Slot-level query count sensor: keeps a running count for each slot S of the number of queries Theo has inferred whose slots equal S.

BEBG

BEBG can be applied only to an existing learned rule. Because EBG is rarely invoked, not many rules are generated, and in general, their usage rate is low. Furthermore, the sensors required to calculate BEBG utility are relatively low-cost. Hence, there is no need to implement adaptive sensing for BEBG.

Sensor Trade-Offs

What does adaptive sensing give us? Using adaptive sensing, ISM is able to reduce D-sensing at the cost of I-sensing. However, it is clear that although ISM can reduce the amount of D-sensing needed, it cannot eliminate it entirely for any query instance. For example, $Cost_{inf}()$ is needed for only some query instances, but $InfStructs$ must be calculated for every query instance so that $InfStructMatches$ may be calculated correctly. What exactly are the sensor tradeoffs?

Figure 6.1 shows the sensing and utility calculation responsibilities of ISM with and without adaptive sensing. Because BEBG utility is calculated so infrequently, it, along with its sensors, have not been associated with either full or adaptive sensing. In general, with adaptive sensing, the sensors whose data can affect the utility of other query instances must always be used.

	Not Adaptive	Adaptive	
	every query	every query	selected queries
U_{cache}	x		x
U_{EBG}	x		x
U_{BEBG}			
$Prob_{stable}$	x		x
$Cost_{inf}$	x		x
$\mathcal{L}[\mathcal{E}(A)]$	x		x
$InfStruct$	x	x	
$\|InfStructMatches\|$	x		x
$\|queries(..)\|$	x		x
$\|queries(slot(..))\|$	x	x	
$Successes(..)$			
$ExpVarHistory(..)$			
$\|ExpVarMatches(..)\|$			
TopLevelQuery		x	
MultipleDependents		x	
UnstableAddress		x	
UnstableDependent		x	

Figure 6.1: ISM With and Without Adaptive Sensing

6.3 Phasic Sensing

ISM uses another strategy to reduce its sensing overhead. In this strategy, sensing is carried out in phases. The basic idea is as follows:

Decision Phase: ISM operates as described above, and attempts to *generalize* some of its management decisions under some conditions – i.e., make decisions that effect sets of problem instances as well as individual problem instances. ISM's generalization strategies are discussed later in this section.

Error-Detection Phase: ISM stops actively managing speedup mechanisms and relies on its previously determined strategy to manage the speedup mechanisms, reducing overhead as well as adaptivity. To ensure *some* adaptivity, however, ISM begins to monitor for *consequential management decision errors*. If enough such errors are detected, ISM falls back to Decision Phase operation.

A *management decision error* is an incorrect management decision that reduces architecture performance relative to optimal speedup mechanism management (with ISM's speedup goals). A *caching decision error* is such an error that ISM has made with the caching speedup mechanism. A *consequential error* is an incorrect management decision that reduces architecture performance relative to Theo's default management strategy. Recall that Theo, by default, caches the values of all problem instances and never invokes EBG or BEBG. If ISM decides to cache a value that should not have been cached, this error is not consequential – relative to default Theo, ISM has not hurt the architecture's performance. On the other hand, if ISM uncaches a value that should have been cached, it has committed a consequential error. More precisely, it has committed a consequential caching error.

The error-detection phase requires less overhead than the decision phase for several reasons. First, ISM does not need to calculate any utilities in this phase. Second, mismanagement detection requires less monitoring than active speedup mechanism management. Finally, in the error-detection phase ISM is concerned with only a subset of the query instances that the decision phase handles – the instances for which ISM may have made consequential errors.

For phasic sensing to be effective, the architecture environment must be "quasi-stable." The dynamics of the environment must change slowly enough that:

- ISM can initiate its error-detection phase
- ISM's management strategy developed from the decision phase remains a reasonable strategy during the error-detection phase

Obviously, quasi-stability is a function of the architecture's domain and environment, and cannot be guaranteed. However, this sensing strategy can take advantage of a quasi-stable environment. If the environment is too dynamic for this strategy, ISM remains in the decision phase. Hence this strategy never compromises ISM's management decisions.

In the rest of this section, phasic sensing is described in detail for each speedup mechanism.

6.3.1 Caching

Decision Phase

In addition to making address-level caching decisions, ISM maintains slot-level data about the percentage and number of times addresses with that slot have been cached by ISM. When ISM enters the error-detection phase, it uses this data to make slot-level caching management decisions.

ISM enters the error-detection phase for caching if ISM's *caching error rate* (caching decision errors / number of query instances) falls below a particular threshold, and Theo has inferred a considerable number of top-level queries. At this point, ISM assumes the environment is stable enough, ISM's caching management decisions are good enough, and ISM has had enough experience that active management is no longer needed.

Error-Detection Phase

During this phase, ISM continues to use the address-level caching decisions developed in the previous phase, but uses an additional, more general slot-level caching scheme. If the caching-percentage and caching-number of a slot exceeds a particular threshold, all addresses having that slot that have not been explicitly uncached are cached.

ISM also monitors the consequential caching errors. If the rate of errors is greater than a threshold, ISM reinitiates decision-phase sensing to cut down on ISM mistakes. There is only one consequential error associated with caching: when ISM uncaches a value that should have been cached. ISM monitors for this situation using the recent answer history datastructure used to calculate $Prob_{stable}$. If the two most recent values are identical, ISM

has made a consequential error. Any query instance for which caching has been turned off is monitored in this way.

For caching, the error-detection phase requires much less overhead than the decision phase. The only sensor needed is the modified $Prob_{stable}$ sensor mentioned above. It is active only for selected query instances – those for which their *whentocache* slot equals *caching.off*, which is easy and inexpensive to determine.

6.3.2 EBG and BEBG

Decision Phase

Because the cost of invoking EBG and BEBG is so high, no generalized decisions regarding their *invocation* are made by ISM – an incorrect decision would be too costly. However, since the sensing, rule application, and rule retrieval costs associated with EBG can be quite costly, ISM has the option of eliminating EBG as a speedup mechanism. As discussed in the EBG utility section, ISM can eliminate specific rules. In addition, ISM can make the following decisions:

Eliminate EBG: if EBG has not been applied in N top-level queries, ISM assumes that the overhead associated with it outweighs its potential benefits, and removes it from consideration as speedup mechanism.

Eliminate all rules: if no learned rules have been successfully applied for N top-level queries, ISM assumes that the rule retrieval and application costs outweigh the benefits of the rules, and sets a flag that prevents Theo from attempting to retrieve any learned rules.

The sensors required by ISM for these decisions are relatively simple and inexpensive.

ISM enters the error-detection phase after the number of top-level queries that ISM has inferred is greater than some threshold. ISM presumes that at this point, ISM has observed Theo's operation for long enough that, had any EBG rule been useful, it would have been generated.

Error-Detection Phase

In this phase, ISM simply continues to use the rules generated by EBG and BEBG in the decision phase. At this point, EBG is no longer considered for use – EBG is essentially turned off. ISM assumes that had EBG been

useful, it would have been invoked in the decision phase. Because Theo, by default, never uses EBG, ISM cannot commit any consequential errors by killing EBG. Consequently, no EBG sensing is needed for this phase, and furthermore, ISM cannot apply EBG in the future. Although this behavior can be very non-optimal in domains with particular characteristics, in natural domains, such as the CAP and Mail-Notification domains used for testing the system, this strategy works well.

However, because the BEBG sensors are very inexpensive, ISM's management of BEBG does not change from the decision phase to the error-detection phase. That is, ISM continues to monitor the variable instantiations of successful expensive rule applications, instantiating expensive variable values if necessary – ISM does not turn BEBG off.

6.4 Adaptive Sensing and Phasic Sensing

Two overhead-reducing strategies have presented. ISM actually uses a combination of both techniques; ISM uses phasic sensing with the following modification: the decision phase uses adaptive sensing. Figure 6.2 shows the sensors needed by ISM for its sensing strategy. The combination of these sensing strategies are summarized for each speedup mechanism in this section.

6.4.1 Caching

Decision Phase

In this phase, ISM fully monitors and computes the caching utility of any query instance Q that satisfies any of the following criteria:

- Q is a top-level query
- Q has multiple distinct dependents
- Q has an unstable dependent
- An expensive rule is being applied, causing Q to be inferred as a subproblem.

To determine when a query instance satisfies any of these criteria, ISM invokes the following sensors on every query instance:

- TopLevelQuery

	Decision Phase		Error-Detection
	every Q	selected Qs	selected Qs
U_{cache}		x	
U_{EBG}		x	
$Prob_{stable}$		x	x
$Cost_{inf}$		x	
$\mathcal{L}[\mathcal{E}(A)]$		x	
$InfStruct$	x		
$\|InfStructMatches\|$		x	
$\|queries(..)\|$		x	
$\|queries(slot(..))\|$	x		
TopLevelQuery	x		
MultipleDependents	x		
UnstableAddress	x		
UnstableDependent	x		
CachingErrorRate		x	x
CachingSlotLevelStats		x	
EBGApplicationRate		x	
RuleApplicationRate		x	x

Figure 6.2: ISM With Adaptive and Phasic Sensing

- MultipleDependents
- UnstableDependent
- ExpensiveRuleApp

If a query instance satisfies one of these criteria, ISM invokes the following sensors on the instance, which are needed to calculate its caching utility:

- U_{cache}
- $Prob_{stable}$
- $Cost_{inf}$
- $\mathcal{L}[\mathcal{E}(A)]$

Whenever ISM considers caching the value of a query instance, two additional sensors are invoked:

- CachingErrorRate
- CachingSlotLevelStats

The first of these sensors enables ISM to determine when its caching decisions are accurate enough that ISM can begin utilizing the error-detection sensing strategy, reducing ISM overhead. The second of these sensors enables ISM to make slot-level caching management decisions.

Error-Detection Phase

During this phase, ISM monitors for consequential caching errors on the query instance values that it had previous decided not to cache. To do this, ISM requires the following sensors:

- $Prob_{stable}$
- CachingErrorRate

Phase Transition Criteria

ISM moves from the adaptive sensing phase to the error-detection sensing phase when $CachingErrorRate < 1/25$ and Theo has inferred more than 20 top-level queries.

Once ISM is in the error-detection sensing phase, it reverts to adaptive sensing if $CachingErrorRate > 1/18$.

These numbers have been empirically determined using the CAP and MN domains described later.

6.4.2 EBG

Decision Phase

ISM invokes the following sensors on every query instance Q to determine whether it should consider applying EBG to Q:

- UnstableAddr
- TopLevelQuery
- $|queries(slot(..))|$

In addition, for ISM to correctly determine EBG utility, ISM needs the ability to determine the inference structures of any query. Hence, ISM also applies this sensor on every query instance:

- $InfStruct$

When ISM decides to calculate the utility of EBG on selected query instances, the following sensors are invoked:

- $|InfStructMatches(..)|$
- $|queries(..)|$

In addition, ISM requires sensors to determine whether EBG should be eliminated as a speedup mechanism. This requires

- EBGApplicationRate

Finally, to determine when EBG sensing should move out of the decision phase, ISM uses

- TopLevelQueryCount

Error-Detection Phase

Once EBG sensing moves out of the decision phase, EBG is no longer considered for application for any query instance. Hence, in this phase, no sensing is required.

Phase Transition Criteria

ISM's EBG management moves out of the decision phase when $TopLevelQueryCount > 20$. This number has been empirically determined.

6.4.3 BEBG

Decision Phase

BEBG is only available as a speedup mechanism when an EBG rule has been applied, which is very infrequent. Hence, sensing costs for BEBG are low, and BEBG does not require adaptive sensing. Hence, the following BEBG sensor is called every time an expensive EBG rule is applied:

- ExpensiveRuleApp

In addition, ISM can eliminate the use of learned rules, if the rule application overhead is deemed too expensive. To gather the data necessary to make such a decision, ISM uses this sensor every time a learned rule is used:

- RuleSuccessRate

Error-Detection Phase and Phase Transition Criteria

Because BEBG sensors are relatively inexpensive, and because they are invoked so infrequently, there is no need to minimize BEBG sensing with an error-detection phase.

6.5 Overhead Management Performance

This section uses data gathered from the CAP domain experiments (described in the "Experimental Results" chapter) to demonstrate the overhead associated with the following three sensing strategies:

- Full sensing: monitor every query instance

Sensor	seconds x 10,000
U_{cache}	7.8
U_{EBG}	12.3
$Prob_{stable}$	3.8
$Cost_{inf}$	7.3
$\mathcal{L}[\mathcal{E}(A)]$	3.3
$InfStruct$	7.7
$\|InfStructMatches\|$	4.2
$\|queries(..)\|$	6.2
$\|queries(slot(..))\|$	6.2
TopLevelQuery	3.5
MultipleDependents	4.0
Unstable Addr	3.5
UnstableDependent	5.2
CachingErrorRate	5.5
CachingSlotLevelStats	6.5
EBGApplicationRate	3.4
RuleApplicationRate	3.4

Figure 6.3: ISM Sensor Time Costs (seconds x 10,000)

- Adaptive sensing
- Adaptive and Phasic sensing

The decision quality and effect on overall system performance of each of these strategies is also examined.

6.5.1 Overhead Calculations

Figure 6.3 shows the time needed to operate each of ISM's sensors 10,000 times.

For the monitors needed by the full-sensing monitoring strategy, ISM generates an overhead of 58.8 seconds for every 10,000 inferences.

In the CAP domain, using the adaptive sensing strategy, ISM invokes its D-sensors only about 6 times for every 1,000 inferences – .6% of the time. If

Strategy	overhead
Full	58.8
Adaptive	30.6
Adaptive/Phasic	between 12.7 and 30.8

Figure 6.4: ISM Overhead per Inference

we conservatively assume D-sensor invocation for 1% of all inferences, ISM generates an overhead of 30.6 seconds for every 10,000 inferences. Adaptive sensing reduces ISM overhead by close to a factor of two.

With both adaptive and phasic sensing, ISM's overhead is reduced even further. Decision-phase sensing generates an overhead of 30.8 seconds for every 10,000 inferences – an overhead similar to that of adaptive sensing. As ISM's sensing strategy moves to the Error-Detection phase, however, overhead falls to 12.7 seconds per 10,000 inferences, even under the assumption that every inference requires the application of the all the sensors associated with the error-detection phase. Hence, for any domain, adaptive/phasic sensing will incur a conservative overhead estimate of between 12.7 and 30.8 seconds per 10,000 inferences. In the CAP domain, once ISM begins using the Error-Detection phase sensing, it never falls back to Decision phase sensing. Hence, as CAP continues to run, ISM's overhead approaches a maximum of only 12.7 seconds per 10,000 inferences.

Figure 6.4 summarizes the overheads associated with each strategy.

6.5.2 Performance

Although ISM's overhead management strategy does reduce ISM overhead significantly, what effect does this strategy have on ISM's decision quality? Does adaptive and phasic sensing actually give ISM enough information to work well?

To examine the effect of overhead management strategies on speedup mechanism management decision quality, ISM was run on the CAP domain (which is described in the Experimental Results chapter of this thesis), with ISM managing caching only. Three overhead management strategies were tested:

- Full Sensing: caching considered for every query instance
- Adaptive

Strategy	Number of Decisions
Full	2,557
Adaptive	556
ISM	349

Figure 6.5: Overhead Management Schemes versus Caching Decision Count

Strategy	Performance (seconds)
Full	180.5
Adaptive	156.1
ISM	122.8

Figure 6.6: Overhead Management Schemes versus Performance

- ISM: Phasic/Adaptive

The results of this experiment are summarized in Figures 6.5 and 6.6, which show the number of management decisions carried out by ISM using each strategy, and the architecture performance (in seconds) using each strategy, respectively.

It is clear that full sensing results in many more decisions than either the ISM or adaptive strategy. This is to be expected: adaptive sensing causes ISM to *implicitly* not cache many query instances that full sensing *explicitly* does not cache. The performance data shows that adaptive sensing results in a more efficient system than full sensing. Hence, the strategy of considering speedup mechanisms for only some query instances is successful: the system, using adaptive sensing, takes only 86 percent of the time needed by the full-sensing system.

The figures also show that ISM's strategy results in fewer decisions than the adaptive strategy. This results from the error-detection phase strategy, as well as ISM's slot-level management decisions. The speedup gained combining adaptive and phasic sensing is dramatic: the system, using ISM's strategy, shows a 33 percent speedup over the system using the full-sensing strategy.

This experiment shows that ISM's management ability is not decreased by the overhead management schemes presented, and that these schemes can have a significant effect on the efficiency of a speedup mechanism manager such as ISM.

Overhead Management Strategy Disadvantages

ISM's overhead management scheme works very well for the CAP domain. Can we expect adaptive and adaptive/phasic sensing to always perform well relative to full sensing? Are there any drawbacks to these strategies?

Relative to full sensing, adaptive sensing can cause ISM to converge more slowly to a good speedup mechanism management strategy. It can take time for the adaptive sensing strategy to realize that a speedup mechanism should have been applied to a particular query instance. For example, consider a query instance Q that has a single stable dependent. With adaptive sensing, ISM will not consider caching Q. However, with full sensing, caching is considered, and may be invoked. Assume that Theo is asked to invoke another query instance that uses Q's value. With full sensing, Q's value may have been cached. With adaptive sensing, Q's value will be cached *after* the second query (because of the MultipleDependents I-sensor) – but time has already been lost relative to the full-sensing scheme. Hence, compared to full sensing, adaptive sensing can cause ISM to find good management strategies more slowly.

This is not usually a problem. As the experimental results show, generally, slow convergence is a transient phenomenon; after an initial "training period," management decision quality using adaptive sensing matches that using full sensing.

Because phasic/adaptive sensing uses adaptive sensing, it has the same potential slow-convergence characteristic. However, there is a more serious problem with phasic/adaptive sensing. This kind of overhead control makes powerful assumptions about the characteristics of the domain. As discussed previously, the domain must be *quasi-stable*. In particular, the fact that EBG can be turned off is potentially a big problem. Domains can be constructed such that EBG is not useful until the system has run, say, 50 top-level queries. With full sensing, EBG can still be invoked. With phasic/adaptive sensing, EBG would not be available.

Although this is a possible scenario, natural domains do not seem to exhibit these kinds of radically-changing characteristics. Hence, in practice, phasic/adaptive sensing results in very good performance.

Chapter 7

Experimental Results

This chapter describes the domains used to test ISM's performance, and presents the experimental results.

7.1 Test Domains

ISM's performance has been tested in two of Theo's domains: the Calendar Apprentice domain and the Mail Notification domain. In this section, these domains are described.

7.1.1 Calendar Apprentice

The Calendar Apprentice, or CAP [Mitchell 94], is an interactive learning apprentice for calendar management built on top of Theo. The goal of CAP is to assist the user in scheduling meetings. From the user's perspective, scheduling a meeting involves determining information such as *when* and *where* the meeting should be held. CAP helps the user determine this information. Specifically, CAP provides advice regarding the following aspects of scheduling meetings:

- What date and time should the meeting be scheduled?
- How long should the meeting last?
- Where should the meeting be held?

CAP learns to give reasonable advice by observing scheduling decisions made by the calendar owner, using them as training examples. Learning generates rules which CAP uses, providing advice tailored to the calendar owner.

The Program

CAP provides an interactive editor that the calendar owner uses to manage meetings. It supports the following actions:

- Create a meeting
- Move a meeting
- Modify some information about a meeting
- Delete a meeting
- Copy a meeting to another time/day
- Add a person to the database
- Confirm a meeting by sending the attendees e-mail

This interface gives CAP the ability to observe the user's actions, essentially generating training examples, and also gives CAP a way to provide advice.

Whenever the user invokes an action, the interface prompts him for information needed to carry out the action. For example, *add-meeting* requires data such as *attendees, date, duration, start-time*, and *location*. At each prompt, CAP presents its best guess as to what the value should be to the user.

CAP and Theo

As mentioned earlier, CAP is implemented on top of Theo. Theo provides the following:

- a frame-based representation for CAP's knowledge base
- inference methods for generating advice to the user
- learning methods for CAP

CAP's knowledge base consists of the current calendar state, approximately 250 people and groups known to the calendar owner (including information relevant to them, such as *position, department, office, e-mail address*), and inference rules for generating advice.

This domain is a useful testbed for ISM for several reasons. Firstly, it is a real-world domain, with real data and realistic distributions of queries and knowledge base updates. Secondly, because CAP must interact with the calendar owner, speed is important; users like fast response.

7.1.2 E-Mail Notification

Often, a user receives such a large volume of e-mail that he cannot read any of it in a timely fashion. This can be a problem if important mail requires immediate attention. In such a situation, it would be useful if the user were notified upon the arrival of important mail.

The E-Mail Notification domain, or MN, consists of an application that notifies the user when "important" mail has arrived; a piece of mail is considered important if its author has a meeting with the user later in the day. This application is useful because often such mail contains information pertinent to the upcoming appointment that the user should be aware of.

MN Domain Theory

This domain is implemented in Theo, and interfaces with CAP's knowledge base, the system clock, and the user's mail queue. To operate, MN requires the following information:

- Subsequent meeting attendees: people with whom the user will meet later today
- Current mail authors: authors of the e-mail in the user's mail queue

Obviously, this information must be constantly updated for MN to work correctly.

To determine the subsequent meeting attendees, MN uses CAP to determine the user's scheduled meetings for the current date. Those meetings that have started prior to the current time are filtered out. The attendees of the remaining meetings are inferred and collected.

To determine the current mail authors, MN simply examines the *from* field of each piece of mail in the user's mail queue.

If any elements in the set of subsequent meeting attendees is also an element in the set of current mail authors, a flag notifying the user is set.

MN's domain theory is:

```
important-mail?(x) -:
  current-mailspool(x, m),
  remaining-meeting-attendees(x, a),
  intersection(m, a, r)
  not(empty(r))
```

```
remaining-meeting-attendees(x, a) :-
  remaining-meetings(x, m),
  attendees(m, a)

remaining-meetings(x, m) :-
  current-time(t),
  current-date(d),
  events(d, m),
  event-type(d, 'meeting),
  start-time(m, s),
  s >= t
```

7.2 Domain Characteristics

The effectiveness of a system such as ISM is heavily influenced by the characteristics of the domain that the system is executing. These characteristics are:

Knowledge Stability: does knowledge base data stay constant, or are values constantly changing?

Query Distribution: are query instances repeated, or are most query instances novel?

Inference Structure Distribution: does a large fraction of inferences share similar inference structures?

Amount of Meta-level Inference: do user-level inferences necessitate very much system-level inference?

Inference Complexity: how complicated are the inferences?

The CAP and MN domains have very different characteristics, and hence are good test domains for ISM. In this section, the characteristics of these domains are discussed, as well as their effect on ISM's performance.

7.2.1 CAP

The CAP domain is highly stable; information in its knowledge base rarely needs to change. For instance, data relevant to each person in the knowledge base, such as position and office, will almost never need to be modified.

Consequently, there is essentially no expected TMS costs associated with caching in this domain, only space costs. Because space costs are presumed to be minor, Theo's default caching strategy – cache everything – can be expected to work quite well. Hence, ISM's caching strategy is not expected to provide much speedup leverage. That is, ISM's caching strategy may increase efficiency, but only due to the second-order effects of more efficient space utilization; the more important truth-maintenance costs associated with caching will not be observed in this domain.

Furthermore, CAP's query distribution is relatively uniform. In general, this means that the inferences required by CAP have relatively widely varying inference structures. EBG and BEBG are useful only for inferences with similar structures, and consequently, EBG and BEBG are not expected to be useful in this domain. Once again, this means that Theo's default invocation strategy for EBG and BEBG (never use either) are very close to optimal, nullifying any potential speedup associated with ISM's intelligent management of these mechanisms.

Because Theo's default use of speedup mechanisms is well-suited for the CAP domain, CAP provides a good test for ISM's sensing control strategy. Even with an effective management strategy, ISM cannot gain a large speedup advantage over Theo. Hence, in this domain, it is imperative that ISM's sensing and control overhead is minimized to ensure that Theo's performance does not degrade. This effect is magnified because CAP's domain characteristics are very stable; the adaptivity of ISM – and the sensing needed to support that adaptivity – is essentially wasted.

On the other hand, the inferences required by CAP are relatively complicated, as evidenced by the rather lengthy times needed for each inference. This implies that CAP requires many system slot inferences. Presumably, ISM's load-time optimization applies to many of these inferences, and the speedup gained from ISM's load-time optimization should be significant.

7.2.2 MN

In many ways, MN's characteristics are diametrically opposed to those of CAP. In the MN domain, information (such as the user mail queue and the user's remaining meetings) is constantly changing. I.e., much of the knowledge is dynamic. In this situation, Theo's default caching strategy works poorly – much cached information will need to be uncached, leading to a high TMS cost. Hence, ISM's adaptive caching strategy should provide a large speedup.

Furthermore, the MN domain specifies that the same inferences are repeatedly requeried – the query distribution is very spiky. Obviously, this means that many inferences have identical structures, making EBG and BEBG potentially very useful. The MN domain, however, contains "expensive" prolog rules, which will result in expensive learned EBG rules, limiting somewhat the utility of EBG. BEBG, however, should provide a dramatic speedup.

One of the interesting characteristics of MN is that data in the knowledge base change at different rates. That is, the level of stability in the knowledge base is subject to change. For example, for a time the user's mail queue may be very stable. Then it may change continually for a period of time. For ISM to maximize system efficiency, it must be able to adapt its management strategies over time, according to these different levels of "dynamicity." Because of this, as well as the dynamic nature of MN and its "spiky" query distribution, MN tests the effectiveness of ISM's management choices and its ability to adapt to changing circumstances.

The inferences in the MN domain, unlike those for CAP, are relatively simple. Furthermore, because of MN's spiky query distribution, system-slot queries are highly repetitive. After these slots are cached, essentially all system slots will be inferred instantaneously. Consequently, the speedup potential of ISM's load-time optimization is very low, limited to only the initial system slot queries. In contrast, consider CAP. Because of CAP's uniform query distribution, Theo must always infer new system slots – slots for which values have not yet been cached. Inferring such slots requires search through the knowledge base – the situation for which ISM's load-time optimization is useful.

7.3 Experimental Results

The following methodology was used for running tests on the CAP and MN domains: a fixed sequence of queries was inferred on several different "systems". These systems consisted of

- Theo
- ISM() – Theo using ISM's load-time optimizations only
- ISM(cache) – Theo using ISM's load-time optimizations and caching control

- ISM(cache, EBG) – Theo using ISM's load-time optimizations with caching and EBG control
- ISM(cache, EBG, BEBG) – Theo using ISM's load-time optimizations with caching, EBG and BEBG control

The experiments were conducted in this incremental fashion to better understand the effects of each speedup mechanism, and to analyze ISM's effectiveness at handling different combinations of mechanisms. The data resulting from these experiments are summarized below.

For the CAP experiments, two sequences of CAP operations were executed. One sequence was recorded from Tom Mitchell's calendar in 1992. It contains 275 operations, consisting of:

- 59 Add-event actions
- 26 Modify-event actions
- 32 Move-event actions
- 105 Copy-event actions
- 53 Delete-event actions

Running these actions results in almost 163,000 Theo inferences, as shown in Figure 7.1.

The other sequence was recorded from Matt Mason's calendar in 1994. It contains 265 operations, consisting of:

- 61 Add-event actions
- 44 Modify-event actions
- 30 Move-event actions
- 96 Copy-event actions
- 15 Delete-event actions
- 15 Add-person actions
- 4 Confirm-event actions

Running these actions results in over 201,000 Theo inferences, as shown in Figure 7.2.

The MN experiments consist of 60 repeated top-level inferences, running three days, and results in about 63,000 Theo inferences, as Figure 7.3 shows. Each top-level query determines if the user has received any important mail, and there are 20 top-level queries per day. Time is updated between queries. The sequence of mail-queue updates, date and time updates, and calendar actions results in a positive response from MN only after queries 2, 5, 9, 10 and 13.

7.3.1 CAP Experiments

In Figures 7.4 and 7.5, Theo is run with and without ISM's load-time optimization technique. ISM() has negligible effect. Why is this the case? During Theo's evolution, performance requirements necessitated compromising some of Theo's design principles. In particular, inference flexibility for some system slots was sacrificed for efficiency. Inference for query instances with these system slots is abridged. Consider a query instance Q with the following address: (a b c...y z). If z is a system slot for which Theo abridges inferences, Q is inferred by inferring the address (y z).

ISM's load-time optimizations operate on a subset of Theo's system slots. Unfortunately, for this experiment, the inferences for these slots have been "pre-optimized" already by Theo, using the abridged inference optimization technique described above. Hence, ISM's load-time optimization is not effective in this situation.

Note an interesting feature in this graphs: sudden large jumps in elapsed time. These jumps result from global garbage-collects performed by Lisp.

One way to test ISM's static optimization technique would be compare its performance to Theo, with Theo's built-in optimization disabled. Unfortunately, this makes Theo unusably slow. Instead, to get some measure of the usefulness of ISM's load-time optimization, ISM is extended to additionally optimize the *whentocache* slot (for only the following experiment). The resulting performance is depicted in Figure 7.6 and Figure 7.7. ISM(), when extended to handle just one additional slot, increases system performance by a factor of about 2.3. Clearly, ISM's load-time optimization technique is very effective for this domain. Note that ISM's static optimization saves space; in this runs, no global garbage collects are needed. Essentially, since ISM() prunes unsuccessful inferences, the space spent otherwise caching the values, explanations, and dependents of these inferences is saved.

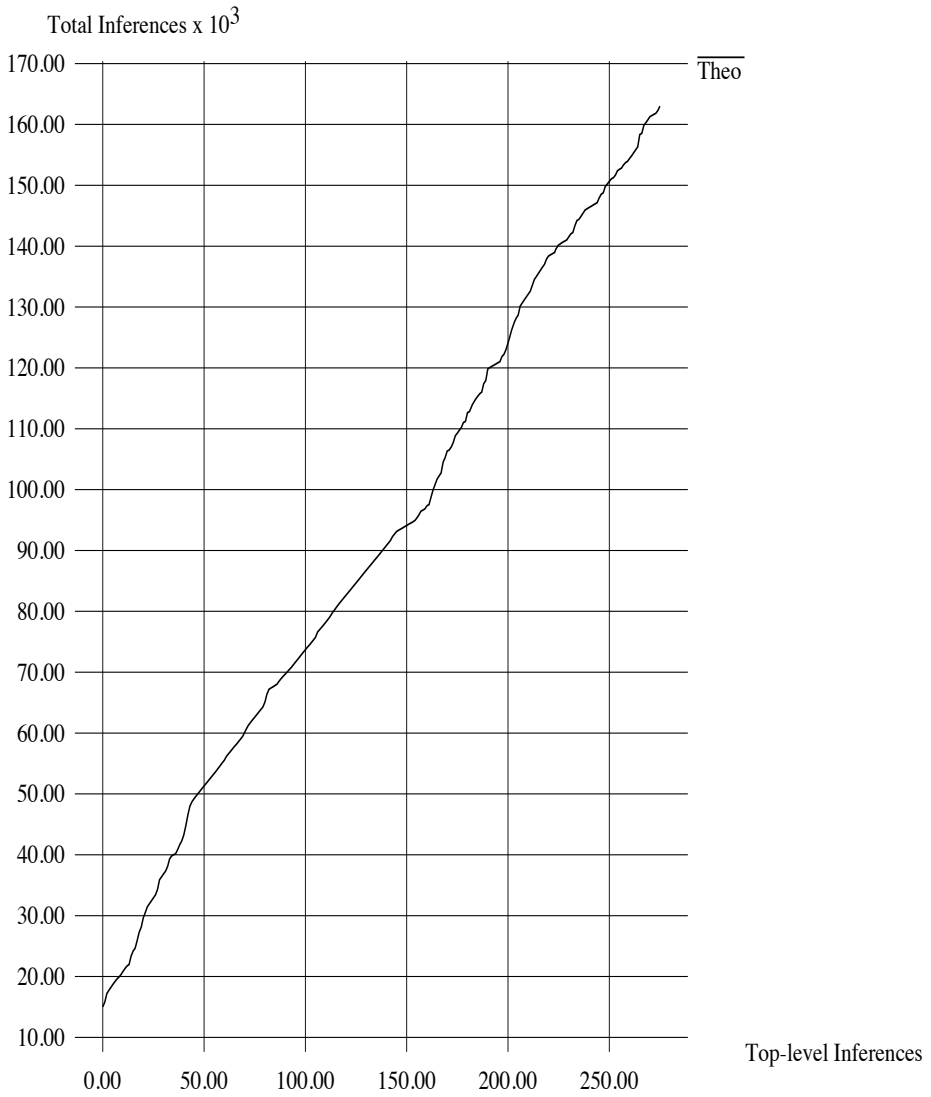

Figure 7.1: Cumulative Number of Theo Inferences versus Number of Top-Level CAP Operations for Mitchell's Calendar

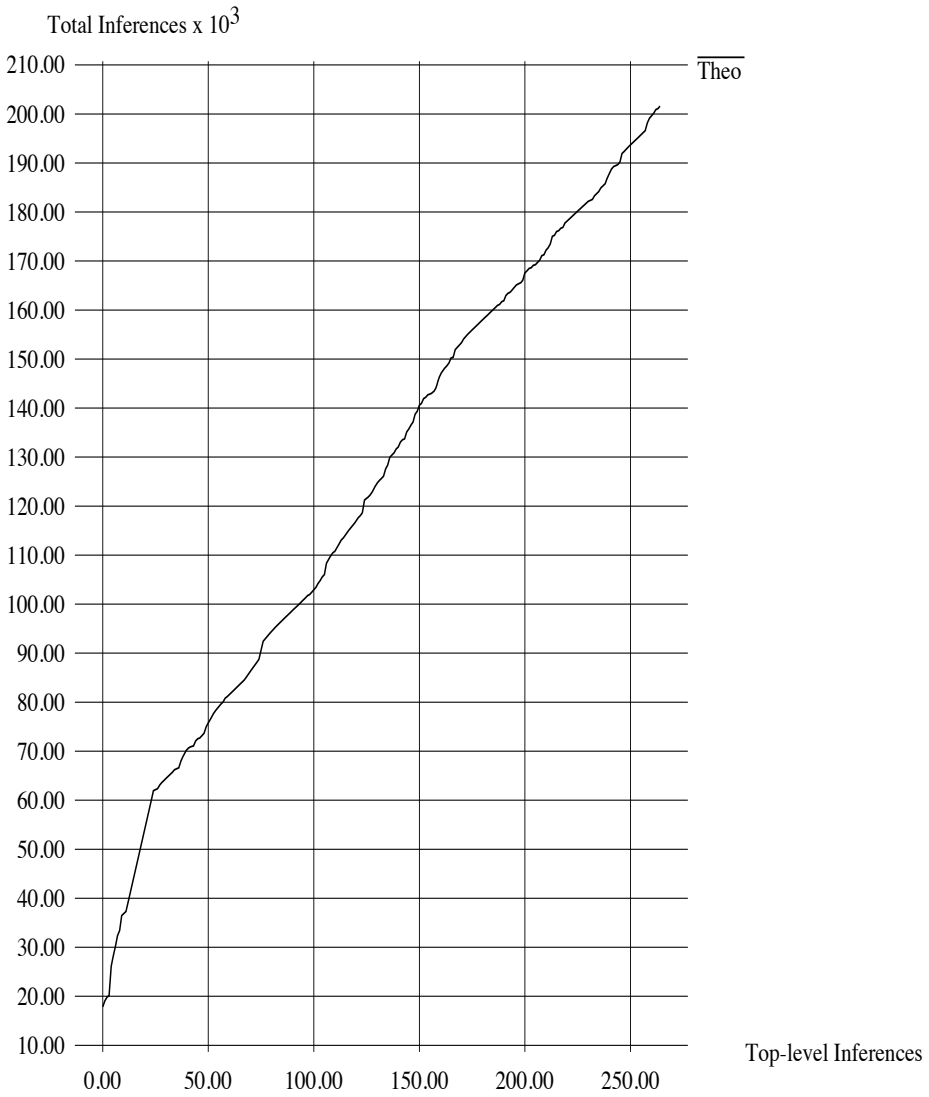

Figure 7.2: Cumulative Number of Theo Inferences versus Number of Top-Level CAP Operations for Mason's Calendar

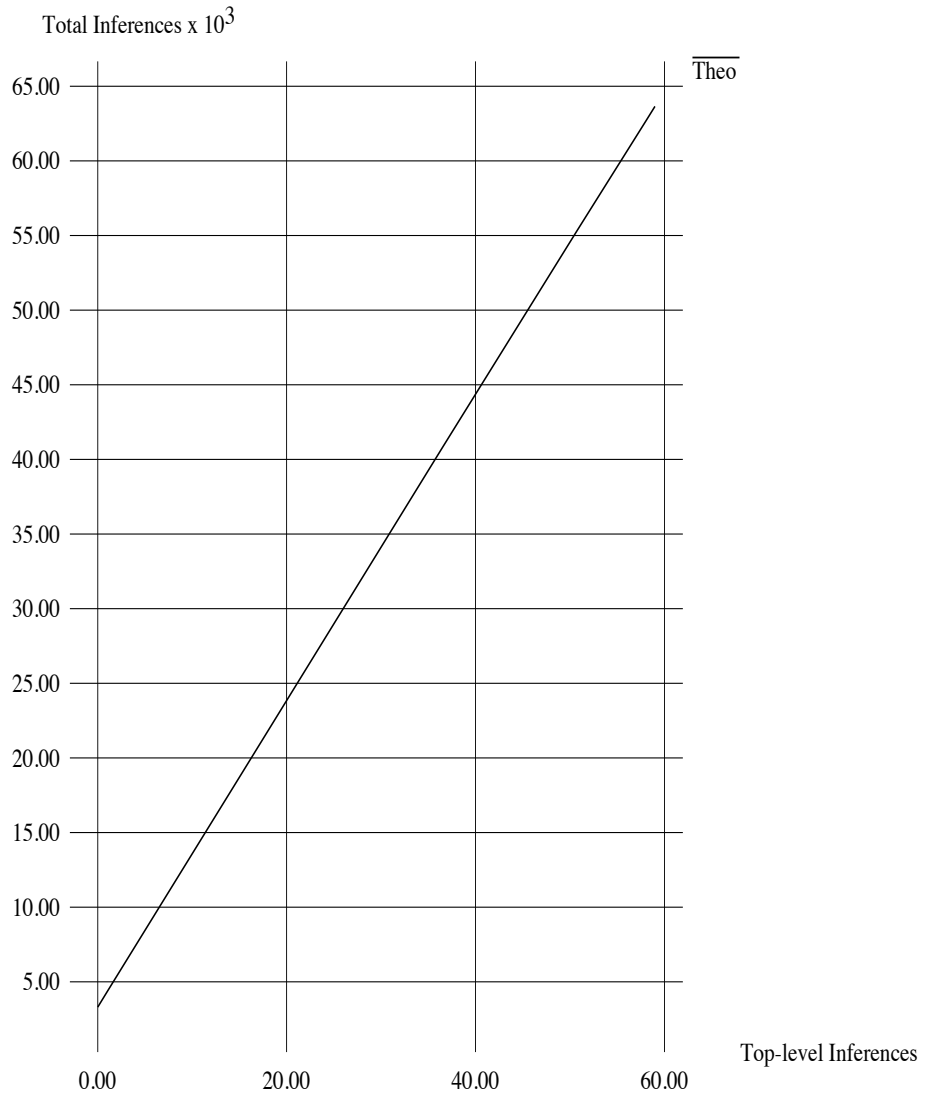

Figure 7.3: Cumulative Number of Theo Inferences versus Number of Top-Level MN Inferences

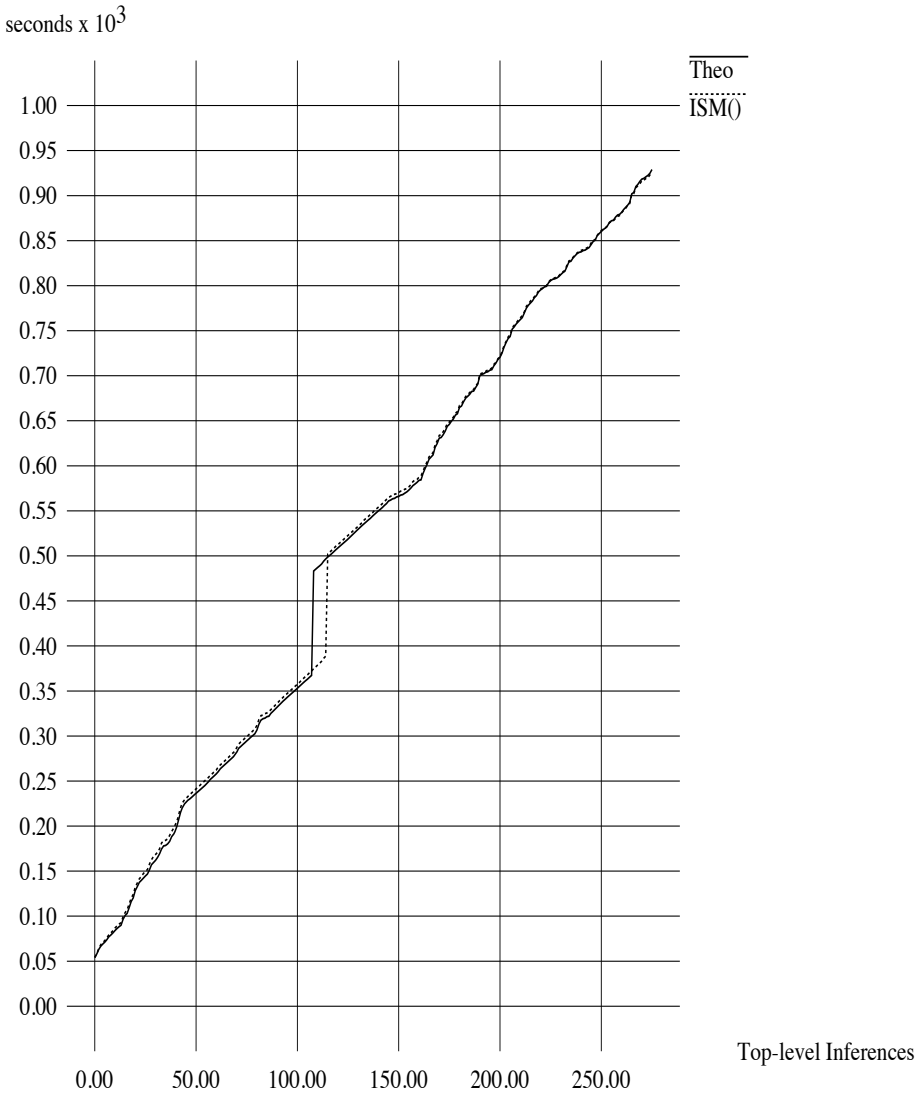

Figure 7.4: Time Elapsed During Top-Level CAP Operation for Mitchell: Theo versus ISM()

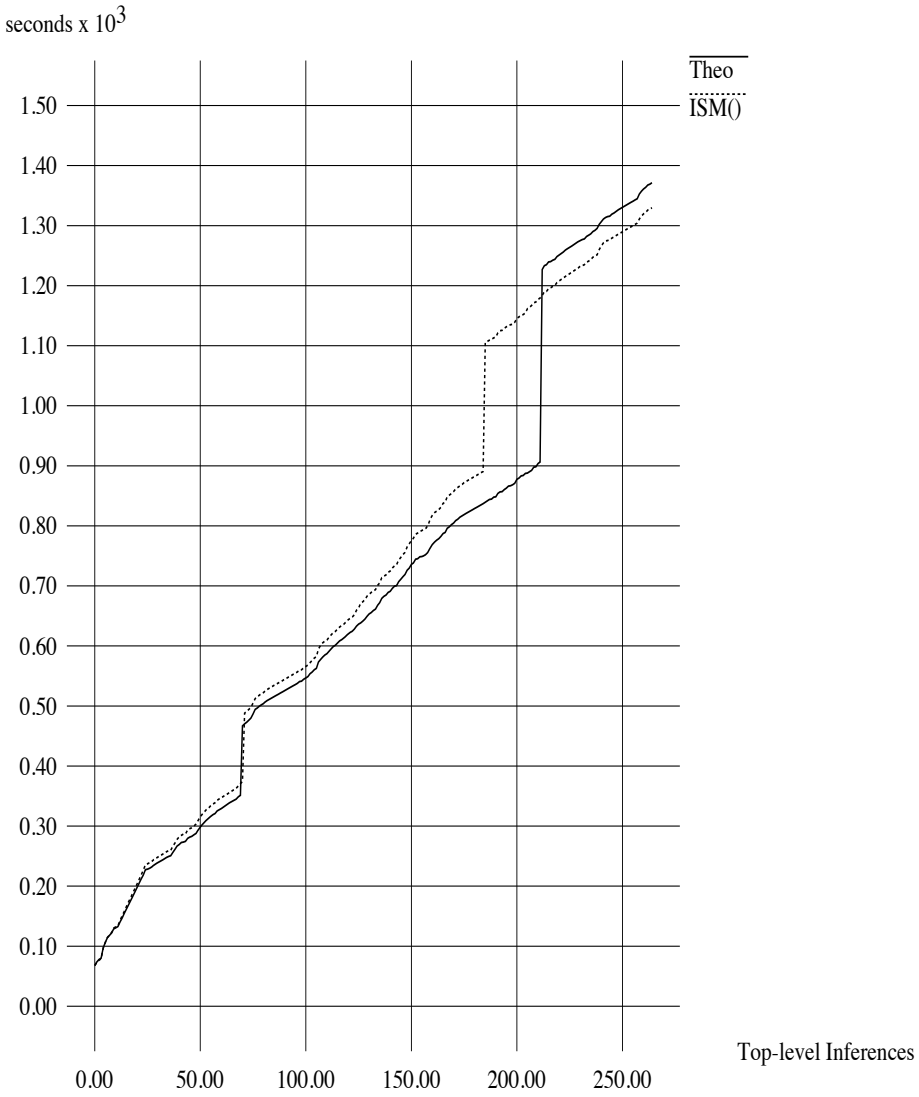

Figure 7.5: Elapsed Time During CAP Operation for Mason: Theo versus ISM()

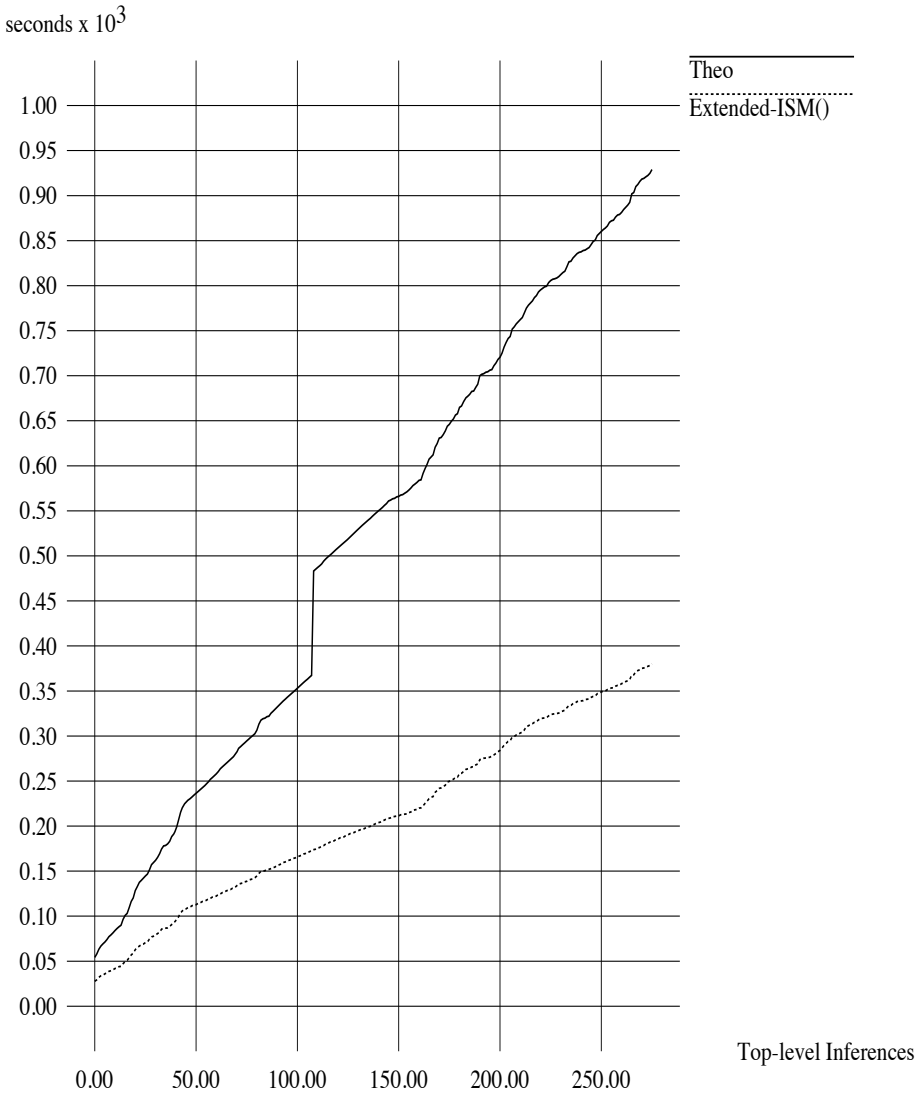

Figure 7.6: Time Elapsed During Mitchell's CAP Operation: Theo versus "Extended" ISM()

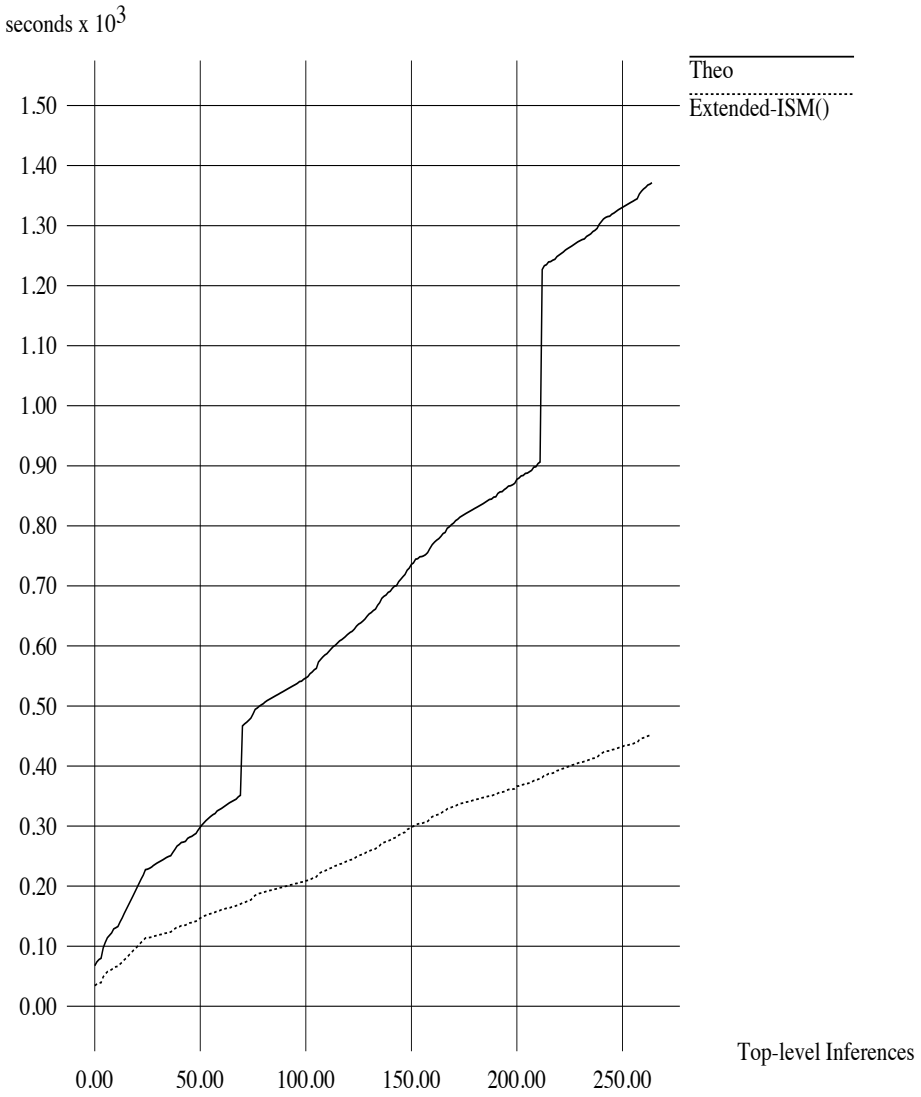

Figure 7.7: Time Elapsed During Mason's CAP Operation: Theo versus "Extended" ISM()

In Figures 7.8 and 7.9, ISM's load-time optimization is combined with run-time caching management. For this test, and those following, ISM does not include the *whentocache* slot in its load-time optimizations – whenever ISM manages caching, there is no need to infer *whentocache* slot values. In this situation, ISM's load-time optimization does not increase Theo efficiency, as Figures 7.4 and 7.5 show; hence, Figures 7.8 and 7.9 essentially depict the isolated effects of ISM's caching strategy. As is evident from the graph, the time savings resulting from ISM's caching strategy is very significant, considering CAP's highly stable knowledge base – speedup is over a factor of 2. This experiment shows that an intelligent caching scheme is very useful even for stable knowledge bases. Also, note that ISM's caching strategy, like its load-time strategy, saves the system global garbage-collect time. This derives from the observation that ISM caches fewer values than Theo. ISM lowers the space requirement of the system.

In Figures 7.10 and 7.11, ISM(cache) is compared with ISM(cache, EBG) and ISM(cache, EBG, BEBG). It is clear that EBG and BEBG do not increase CAP performance – the ISM(cache), ISM(cache, EBG) and ISM(cache, EBG, BEBG) curves are virtually identical. However, this was an expected consequence of CAP's domain characteristics. Indeed, in this test, ISM never invokes EBG or BEBG. Interestingly, despite this, the performance curves between ISM(cache, EBG, BEBG), ISM(cache, EBG), and ISM(cache) do not differ significantly. This demonstrates that ISM's sensing control is effective enough that the additional control and sensing needed by ISM to manage EBG and BEBG are minor.

It would be useful to see how ISM's dynamic and static optimization techniques work together. Unfortunately, as stated earlier, this is impossible with Theo, since ISM's static analysis optimizes the same slots for which Theo has built-in optimizations. Hence, to test interactions between ISM's static and dynamic components, a variant of Theo called *FlexTheo* has been implemented. FlexTheo disables Theo's built-in *methods inferencing* optimization. Hence, although FlexTheo's *methods* are more configurable than Theo's, FlexTheo is less efficient than Theo. FlexTheo's performance is compared against Theo's performance in Figure 7.12. Using FlexTheo, we can now determine how well ISM's static and dynamic optimizations work together. In this experiment, ISM is embedded in FlexTheo. Then FlexTheo's performance is compared with and without ISM on Mitchell's CAP data. The results are shown in Figure 7.13. Note that ISM() speedup up FlexTheo by over a factor of two. With ISM's run-time speedup mechanism management added, efficiency is increased about another factor of

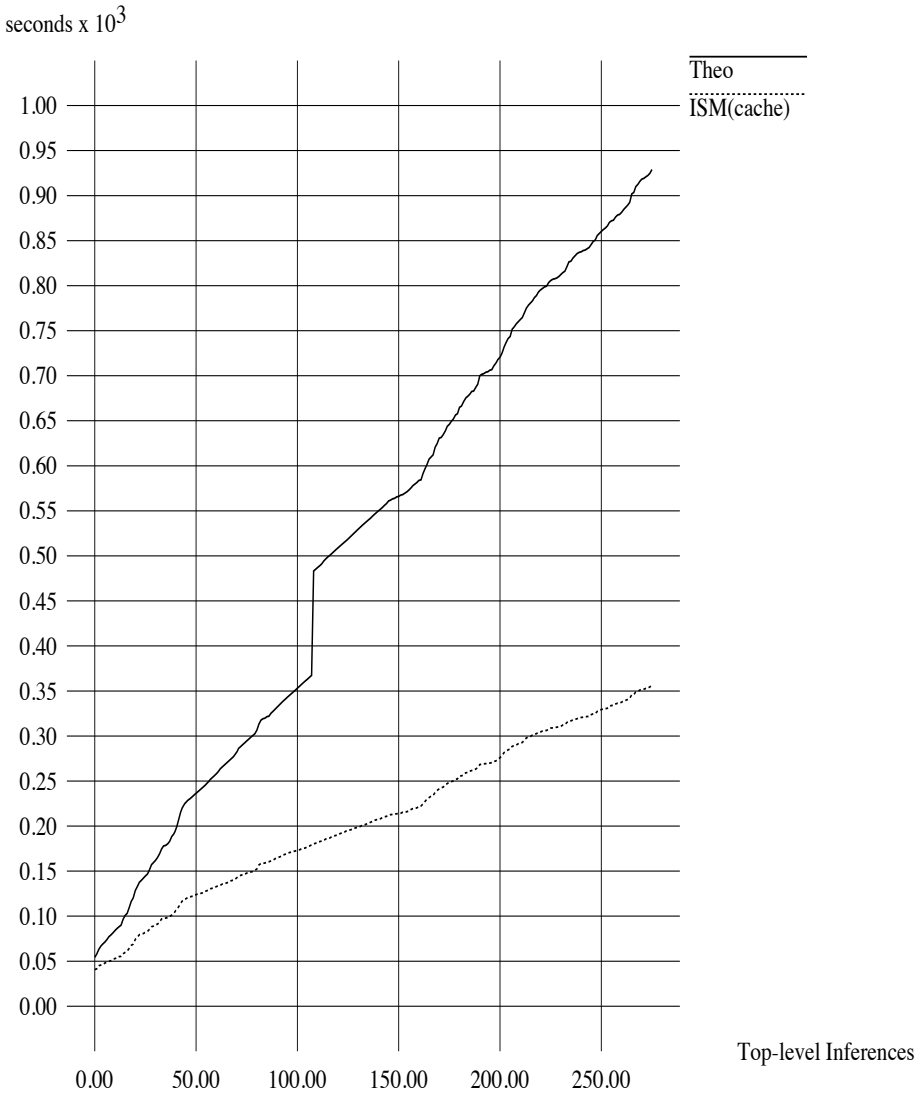

Figure 7.8: Elapsed Time During Top-Level CAP Operation for Mitchell: Theo versus ISM(cache)

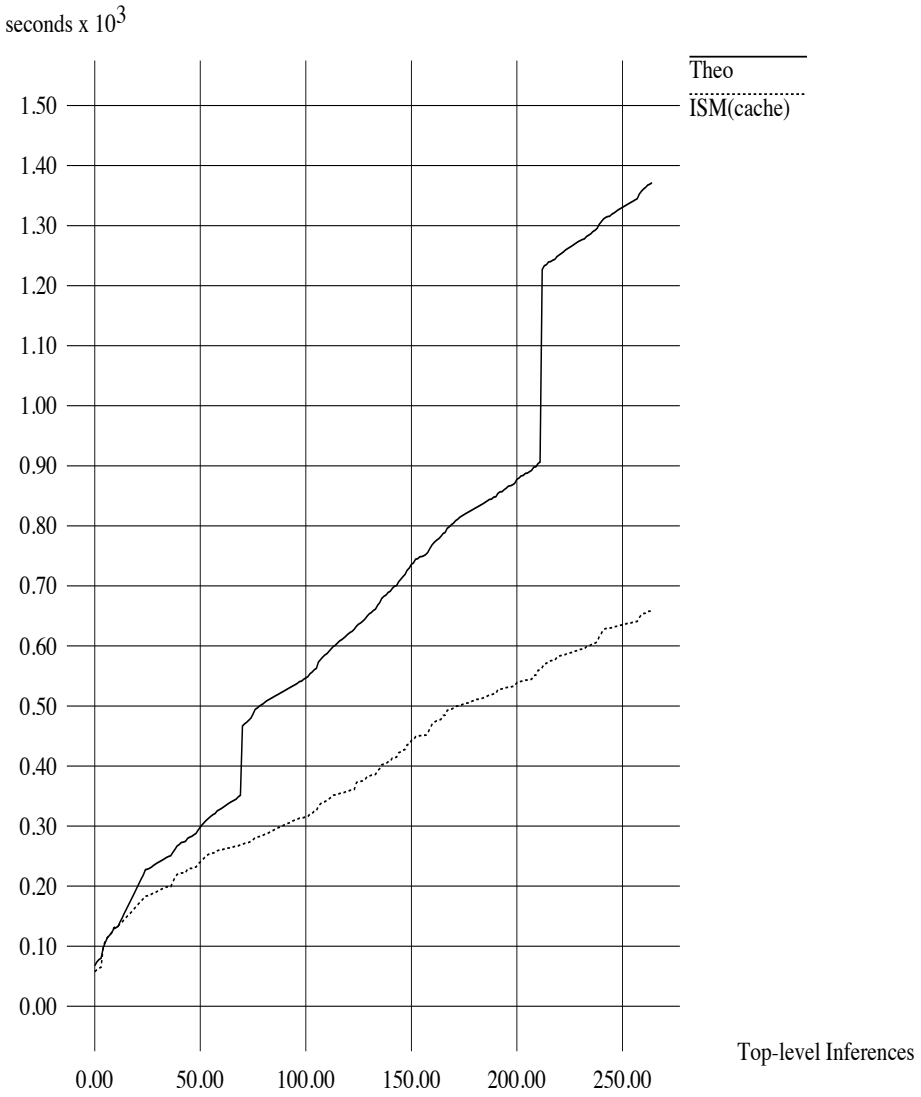

Figure 7.9: Elapsed Time During Top-Level CAP Operation for Mason: Theo versus ISM(cache)

two. Overall, ISM speeds up FlexTheo by a factor of four. This data shows that ISM's static and dynamic optimizations can be effective *in combination* – they complement each other. Note that this experiment consists of only about 90 top-level CAP operations. FlexTheo requires much more memory than Theo. Because of this, it was possible to run FlexTheo for only a limited time.

It would be interesting to pit ISM's static component against ISM's dynamic component using FlexTheo. Unfortunately, ISM's sensing control is designed for Theo, not FlexTheo. Consequently, ISM's dynamic component, if isolated, does not perform well on FlexTheo. Essentially FlexTheo violates some of the assumptions that ISM uses to manage overhead. ISM's static component prunes the inference paths that violate these assumptions. Hence ISM's run-time optimization works well with the load-time optimization, but works poorly on its own, actually reducing FlexTheo's efficiency. This presents more evidence that ISM's run-time and load-time optimizations are synergistic.

The CAP domain testing shows that:

- ISM's load-time optimizations have a potentially very large effect. In this case, when applied to only a small number of Theo's system slots, this technique results in a speedup factor of over 2, as Figures 7.10 and 7.11 show.

- ISM's run-time management of caching leads to a significant speedup even in a knowledge-base with very stable data. Because the data is stable, the major cost associated with caching – TMS costs – are not an issue. However, ISM's handling of caching reduces data storage costs to such an extent that performance increases by a factor of 2.

- ISM's caching sensing control is effective. The sensing needed for caching is efficient enough that ISM realizes a significant net performance increase, even in a domain whose characteristics are not well-suited toward effective caching management.

- ISM's EBG and BEBG sensing control is effective. Despite the ineffectiveness of both these mechanisms for this domain, the additional sensing costs incurred by EBG and BEBG management are minor.

- ISM's dynamic and static optimizations can work well together. Using FlexTheo as a testbed, ISM's static and dynamic techniques each

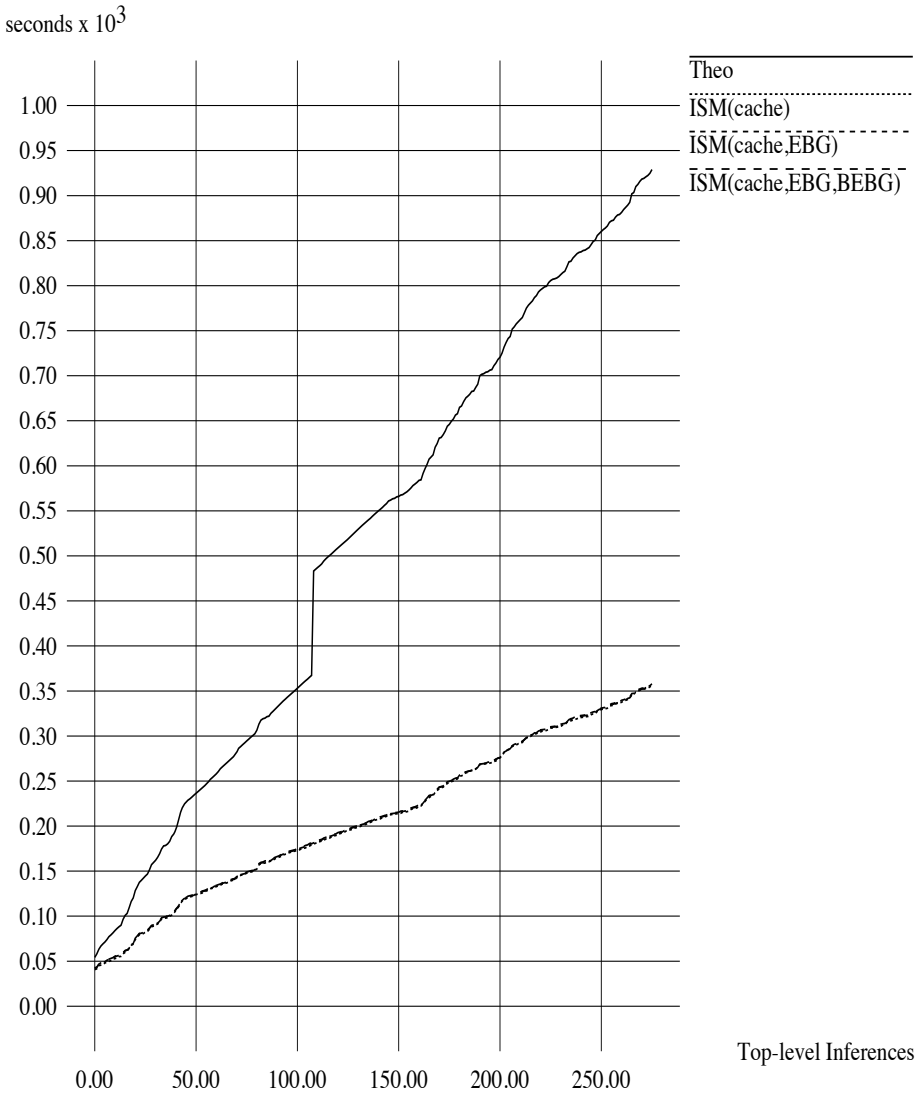

Figure 7.10: Elapsed Time During CAP Operation for Mitchell: ISM(cache) versus ISM(cache, EBG) and ISM(cache, EBG, BEBG)

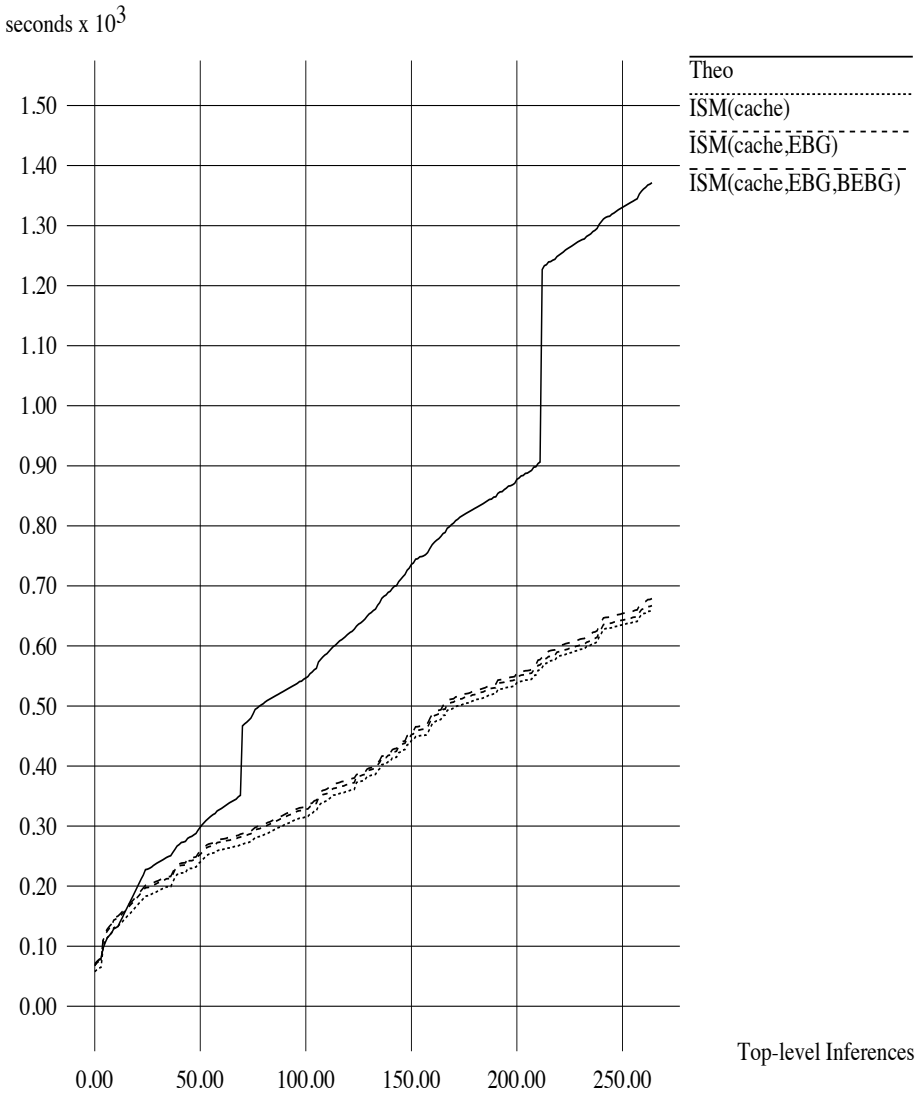

Figure 7.11: Elapsed Time During Top-Level CAP Operation for Mason: ISM(cache) versus ISM(cache, EBG) and ISM(cache, EBG, BEBG)

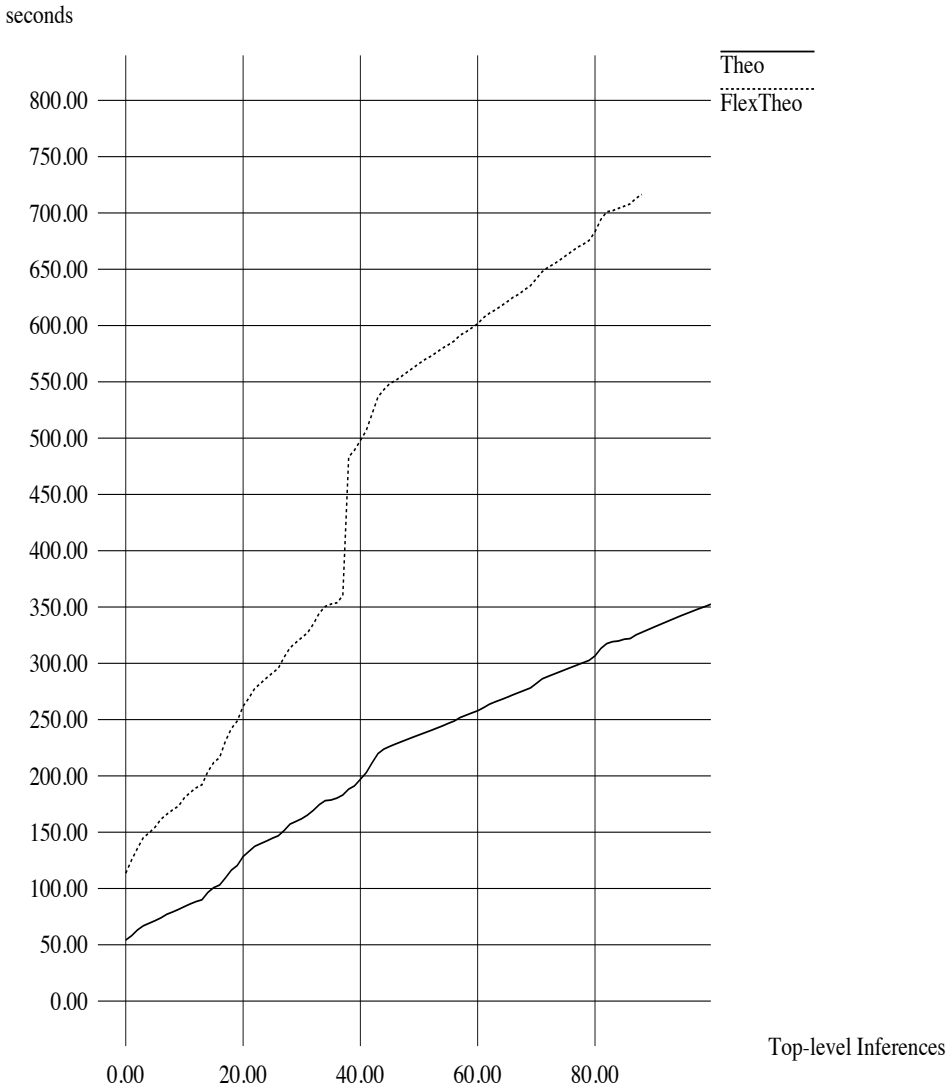

Figure 7.12: Elapsed Time During Top-Level CAP Operation for Mitchell: FlexTheo verses Theo

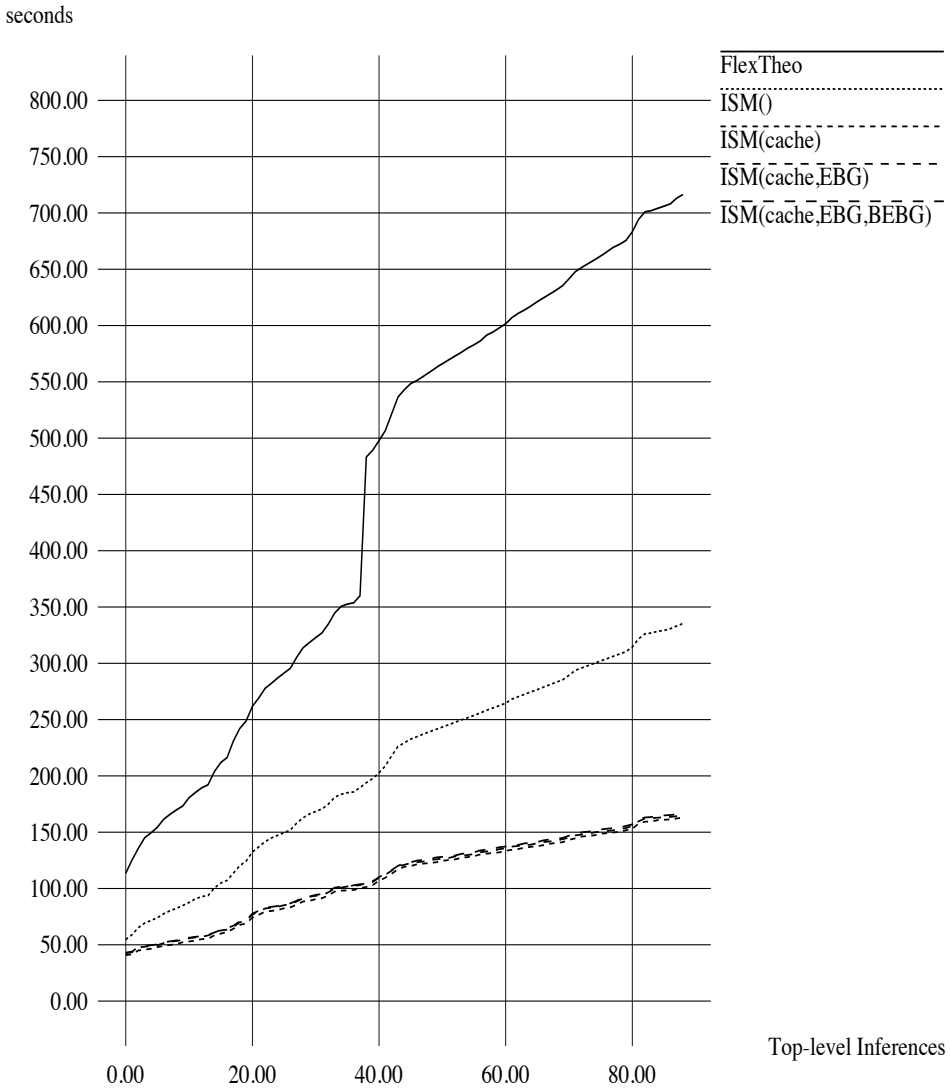

Figure 7.13: Elapsed Time During CAP Operation for Mitchell: FlexTheo with and without ISM

independently increase system efficiency by a factor of two, giving a total speedup of a factor of four.

7.3.2 MN Experiments

The set of experiments run using the CAP domain were also run for the MN domain. That is, ISM(), ISM(cache), ISM(cache, EBG) and ISM(cache, EBG, BEBG) each executed a fixed sequence of MN inferences.

Figure 7.14 displays the effects of ISM's caching strategy in the dynamic MN environment. This figure shows that, initially, ISM is less efficient than Theo – ISM is determining an appropriate caching strategy. At about the ninth top-level query, however, ISM's sensors have collected enough data for ISM to initiate a good caching strategy, and from that point on, its caching strategy increases system efficiency by a factor of about 1.4.

This graph demonstrates an interesting phenomenon. Why is ISM's performance so jittery after the tenth top-level inference? Much of this spiky behavior is a consequence of some inherent limitations to sensor accuracy. Consider ISM's sensor that estimates the stability of a query instance Q. Recall that this sensor operates by keeping a short history of the values of Q, comparing consecutive values to give a probability that Q is stable. If the values of a query instance are cached, this sensor is accurate. However, if Q's values are not cached, under some circumstances the sensor is inaccurate. Consider the following situation: assume Theo continually infers the value of Q =(box area) by multiplying the values of (box width) and (box length). Assume (box length) oscillates between two values, 2 and 4, and (box width) oscillates at the same frequency between 2 and 1. This means (box area) always equals 4, even though its *contributors* are unstable. If Q is cached, this instability is noticed, because when Q's contributor values change, Theo's TMS uncaches Q's value. However, if Q is not cached, its contributor instability is not sensed. This sensor monitors a sequence of 4s, causing ISM to assume that Q's contributors are stable. This problem stems from the fact that ISM *estimates* Q's contributor stability using Q's stability.

The MN domain demonstrates this sensor limitation. This limitation induces the following repeating ISM behavior:

- A cached address is determined to be unstable.
- The unstable address is uncached by ISM, resulting in speedup.

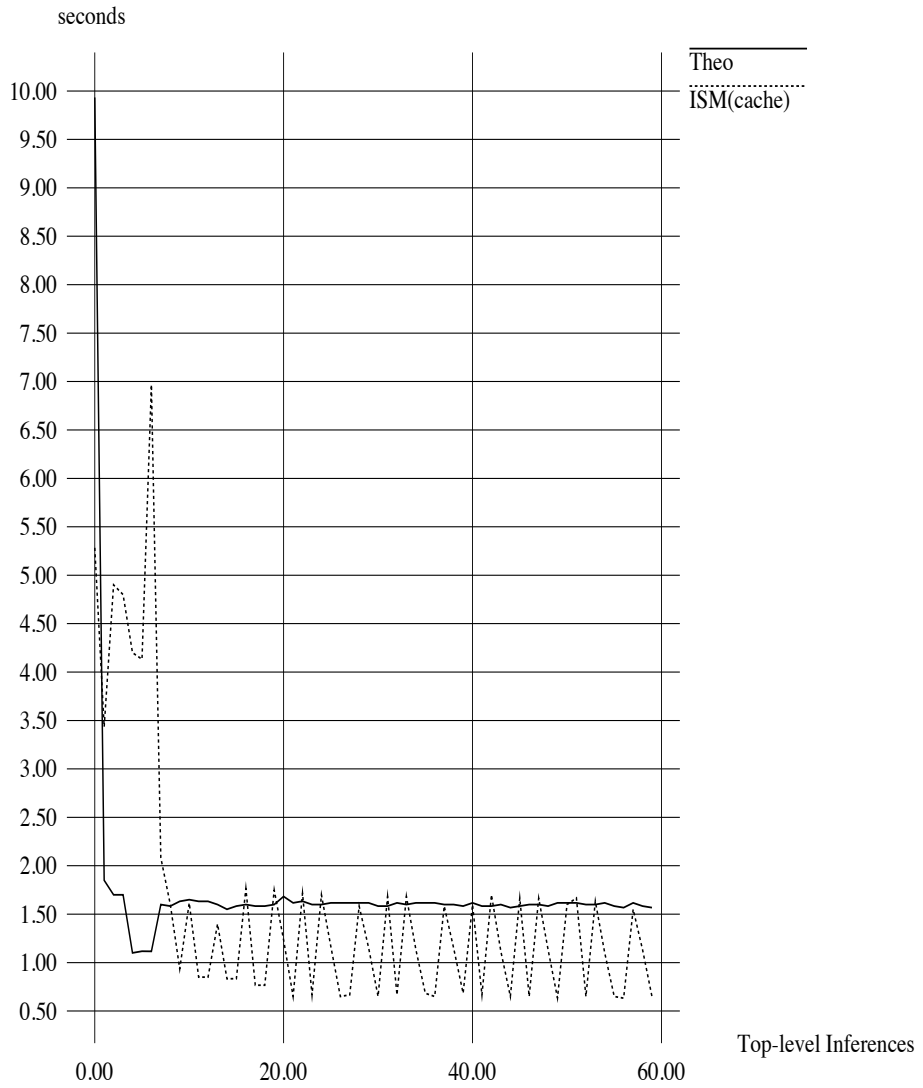

Figure 7.14: Time per Top-Level MN Inference: Theo versus ISM(cache)

- This address has repeating subsequent values, causing ISM to consider the address stable.
- ISM caches the address, resulting in slowdown.
- ISM once again realizes that the address is unstable.

This kind of "sensor aliasing" makes ISM continually oscillate between caching and uncaching a query instance. Because uncaching is actually the optimal management strategy, ISM's performance suffers.

ISM's oscillatory behavior is an artifact of the MN domain. To demonstrate ISM's performance without this sensor aliasing effect, ISM has been tested in a synthetic domain. This "family" domain consists of the following relationships:

```
daughters(x, d) :- children(x, d), female(d)
children(x, c) :- parents(c, x)
parents(x, p) :- father(x, p)
parents(x, p) :- mother(x, p)
```

The following query distribution and knowledge base update patterns were used:

- (tom daughters) was continuously requeried at the top-level.
- After even queries, the following knowledge base modifications where made:
  1. (meghan father) = tom
  2. (shannon father) = *novalue*
- After odd queries, the following knowledge base modifications where made:
  1. (meghan father) = *novalue*
  2. (shannon father) = tom

This pattern shares MN's instability characteristic, but avoids the sensor aliasing problem evident in MN. Figure 7.15 shows Theo and ISM(cache) performance for this domain. As is evident, ISM's oscillation has been eliminated, resulting in a better ISM performance, with a steady-state speedup factor of about 1.6, as shown in Figure 7.16. The spikes in this figure

result from Lisp ephemeral garbage collection. Notice that, because ISM caches fewer values that subsequently need to be discarded by Theo's truth-maintenance, ISM requires fewer garbage-collects than Theo – about half as many. Note that garbage collection accounts for some of the ISM(cache) spikes in the MN domain test. However, garbage collection is not evident in any of the other tests. This results from the fact that in the other tests, inferences are long enough that garbage collection is needed for every top-level inference. Because garbage collection are fairly constant, no garbage collection spikes are present.

In Figure 7.17, ISM manages EBG as well as caching. Because MN's domain theory results in expensive rules, ISM's management of EBG should not have a significant effect on efficiency. The data in Figure 7.17 supports this prediction. As is the case for caching, ISM spends the first eight top-level queries deciding on a strategy. At this point, ISM generates a rule for the (repeated) top-level query, resulting in faster query responses. At the twentieth top-level query, cheap sensing is initiated, further increasing system speed. The sensor aliasing discussed previously is also evident in this experiment; ISM's performance oscillates. Although EBG results in an expensive learned rule, the addition of EBG management to caching nevertheless increases MN performance slightly, compared to only caching management.

Figure 7.18 depicts ISM(cache, EBG, BEBG) performance. In this test, as before, EBG is invoked during the eighth top-level inference, generating a rule for the top-level query. MN's spiky query distribution ensures that the rule is applicable. Unfortunately, because of the expensive nature of the rule, the rule does not actually result in system speedup. However, during the application of the expensive rule over the next five top-level inferences, ISM realizes that the rule can be specialized via expensive variable instantiation while still maintaining applicability. Hence, ISM applies BEBG to the expensive rule at the thirteenth query, reducing its application cost. At this point, the rule's cost becomes bounded and performance skyrockets.

Figure 7.18 depicts a performance slowdown around the twentieth top-level inference. This is due to dynamics in the knowledge base. Specifically, at this point in the experiment, data changes, making the previously learned bounded-cost rule useless. I.e., the variable instantiation carried out by BEBG has restricted the generality of the rule to such an extent that it can no longer be used. Hence, top-level query instance twenty is inferred using the more general, expensive EBG rule. However, MN's domain characteristics cause ISM to later *reinvoke* BEBG at query instance 21, generating a

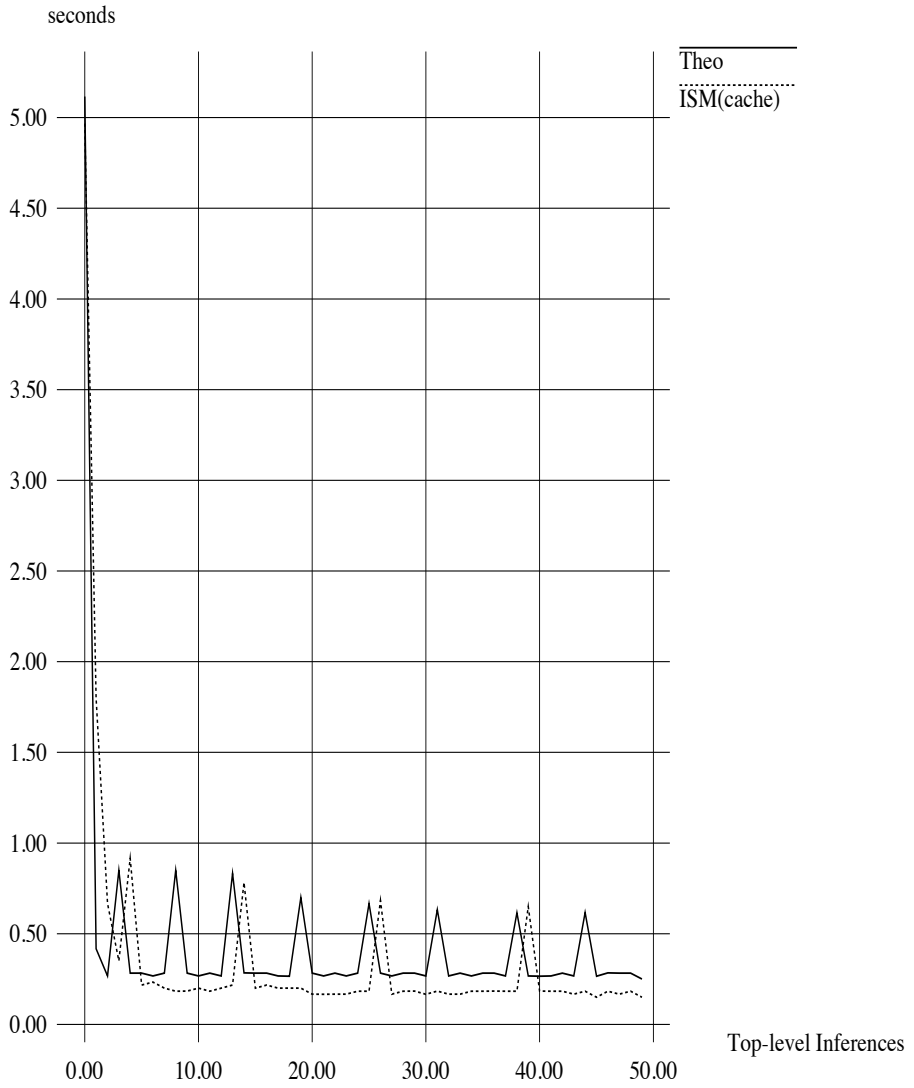

Figure 7.15: Time per Top-Level Inference in a Dynamic Domain: Theo versus ISM(cache)

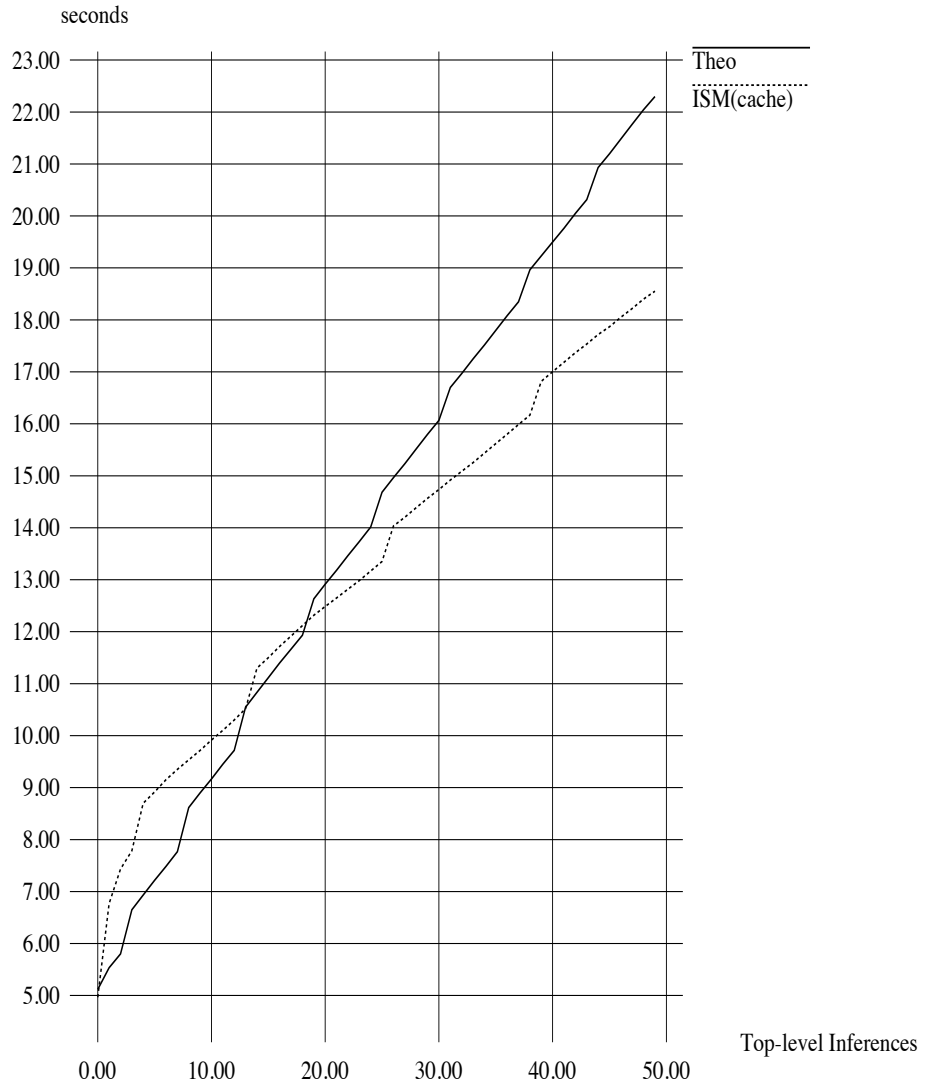

Figure 7.16: Cumulative Inference Time in a Dynamic Domain: Theo versus ISM(cache)

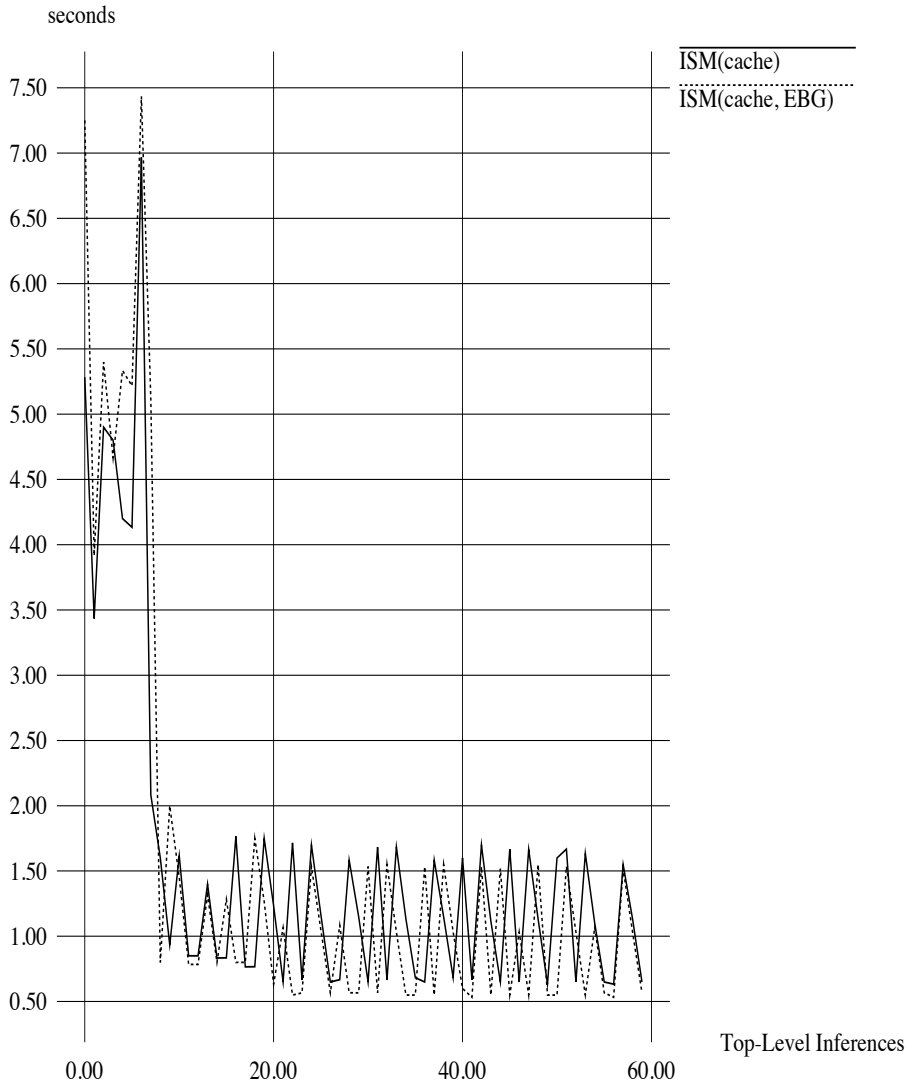

Figure 7.17: Time per Top-Level MN Operation: ISM(cache, EBG) versus ISM(cache)

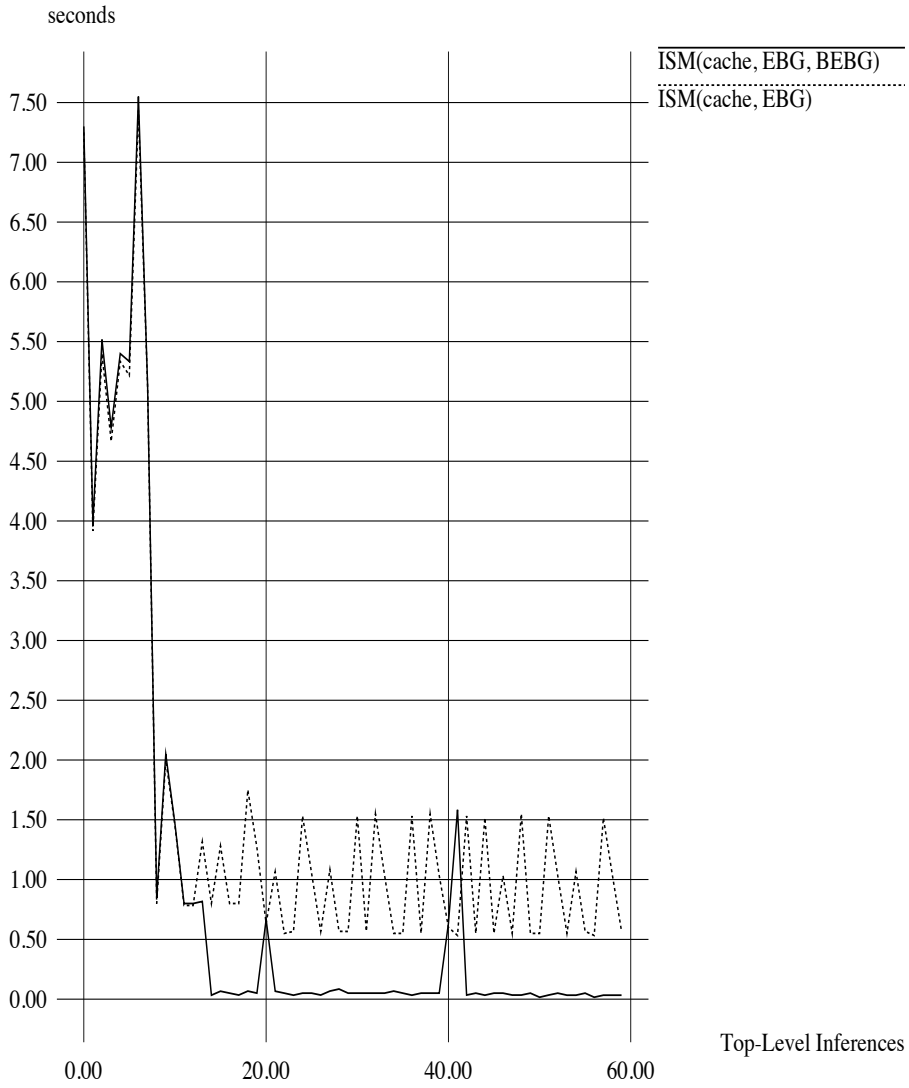

Figure 7.18: Time per Top-Level MN Operation: ISM(cache, EBG, BEBG) versus ISM(cache, EBG)

new bounded-cost rule, applicable to the new state of the knowledge base. At this point performance increases to its previous level.

The fortieth top-level inference in Figure 7.18 shows the same phenomenon. This spike is larger due to some concurrent garbage collection activity. Also note that this spike is wider – ISM needs more time to decide to reinvoke BEBG. Why is this? At query 40, ISM has collected more rule application data than at query 20. Because of the *long-term* nature of the EBG/BEBG sensors, ISM is more likely to view the transient at query 40 as "noisy data" than at query 20. Essentially, as ISM gets older, it requires more evidence of environment dynamics before it adapts. In any case, however, both these transients demonstrate the adaptive quality of ISM's BEBG management strategy.

The MN experiments demonstrate the effectiveness of ISM's speedup mechanism management. In this domain, unlike CAP, all three speedup mechanisms are useful. These experiments show that ISM manages them quite well. As ISM is given more mechanisms to manage, performance increases. In Figure 7.19, ISM's cumulative time savings is shown when all three speedup mechanisms are utilized.

7.3.3 Simulated Domain Experiments

Part of the attractiveness of ISM is its adaptivity. ISM is designed to be more flexible than any fixed speedup mechanism management strategy. The MN experiments have demonstrated a portion of ISM's flexibility – ISM detects when bounded-cost rules become useless due to environment changes, and reacts appropriately. Unfortunately, CAP and MN do not test other aspects of ISM adaptivity. Consequently, simulated domains have been constructed which address these issues.

To test caching adaptivity, the synthetic "family" domain relationships were again used, but with different usage patterns. This domain has the following query distribution and knowledge base update patterns:

- (tom daughters) is continuously requeried at the top-level.
- The following knowledge base update pattern is repeated:
    1. For 10 queries, the knowledge base is stable with (meghan father) = tom. No knowledge base modifications are made.
    2. For 10 queries, the knowledge base is unstable, using the same knowledge base update pattern as was used in Figure 7.15.

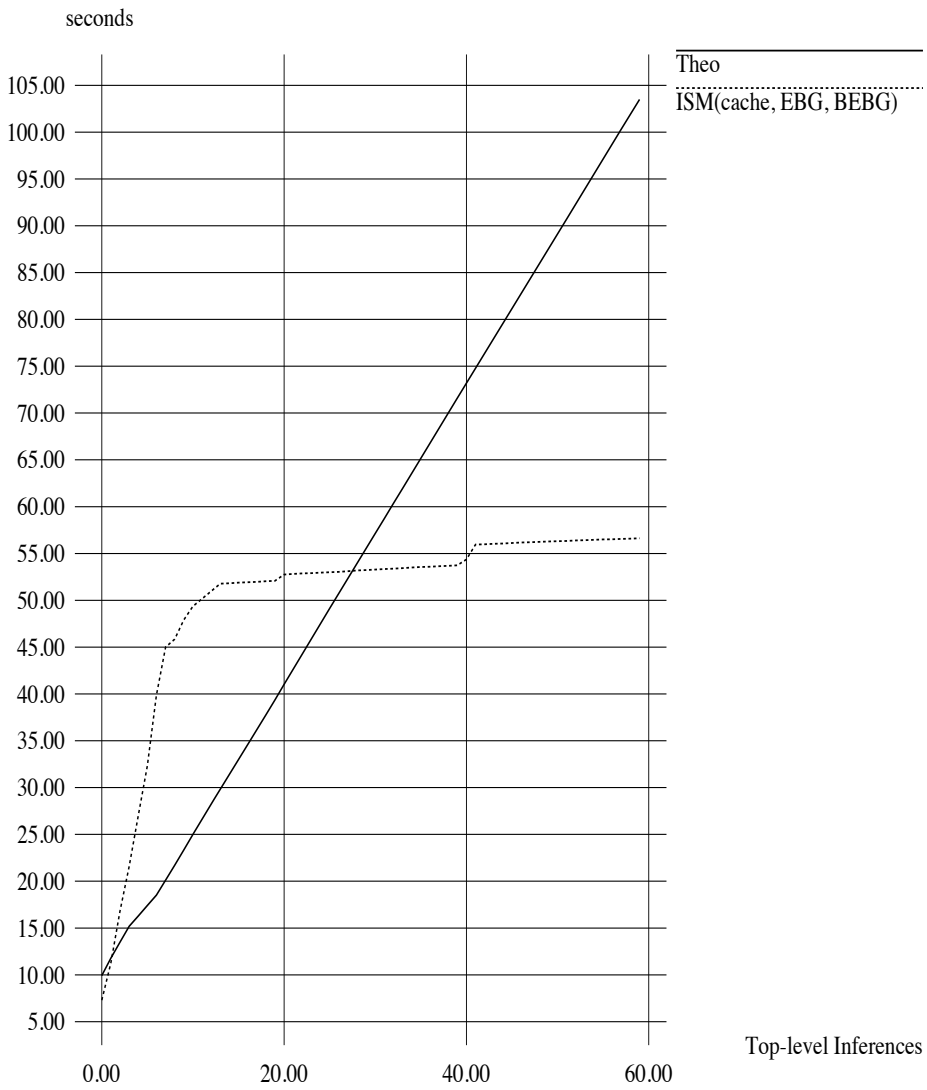

Figure 7.19: MN Elapsed Time: Theo versus ISM(cache, EBG, BEBG)

Hence, this test has the following characteristics:

- Recurring top-level query instance.
- Alternating periods of knowledge stability and instability.

An optimal caching strategy for the top-level query instance of this domain would be: cache first ten queries, uncache next ten, cache next ten, etc. In Figure 7.20, ISM's caching performance is shown. ISM's caching strategy always converges to the optimal strategy during each 10-query cycle. This figure, however, shows that ISM's caching management decisions are slightly delayed relative to the optimal strategy. Obviously, this delay is caused by the fact that ISM cannot detect environment dynamics instantaneously. It is clear from this figure, however, that ISM is able to react very quickly to environment dynamics – in this experiment, ISM adapts consistently within two top-level inferences. This experiment is another demonstration of ISM's ability to adapt quickly to dynamic conditions.

7.4 Summary

The experimental results in this chapter show that

- ISM's speedup mechanisms management strategy is effective across domains with widely varying characteristics. ISM utilizes speedup mechanisms with enough facility to significantly increase Theo efficiency in both the CAP and MN applications. Moreover, ISM does not invoke speedup mechanisms when they are inappropriate. This means that ISM's utility calculations and estimates are effective.
- ISM's dynamic and static optimizations operation work well in combination. They optimize different aspects of the system; hence, combining techniques is effective.
- ISM is able to adapt to *changes* in the environment, showing that ISM's speedup mechanism management strategy is more flexible than any fixed strategy. The MN and synthetic domain tests demonstrate that ISM responds to knowledge base dynamics by modifying its management strategy in a timely and effective fashion.
- ISM's load-time optimizations are effective. Even when applied to only a fraction of Theo's system slots, CAP performance increases by more than a factor of two.

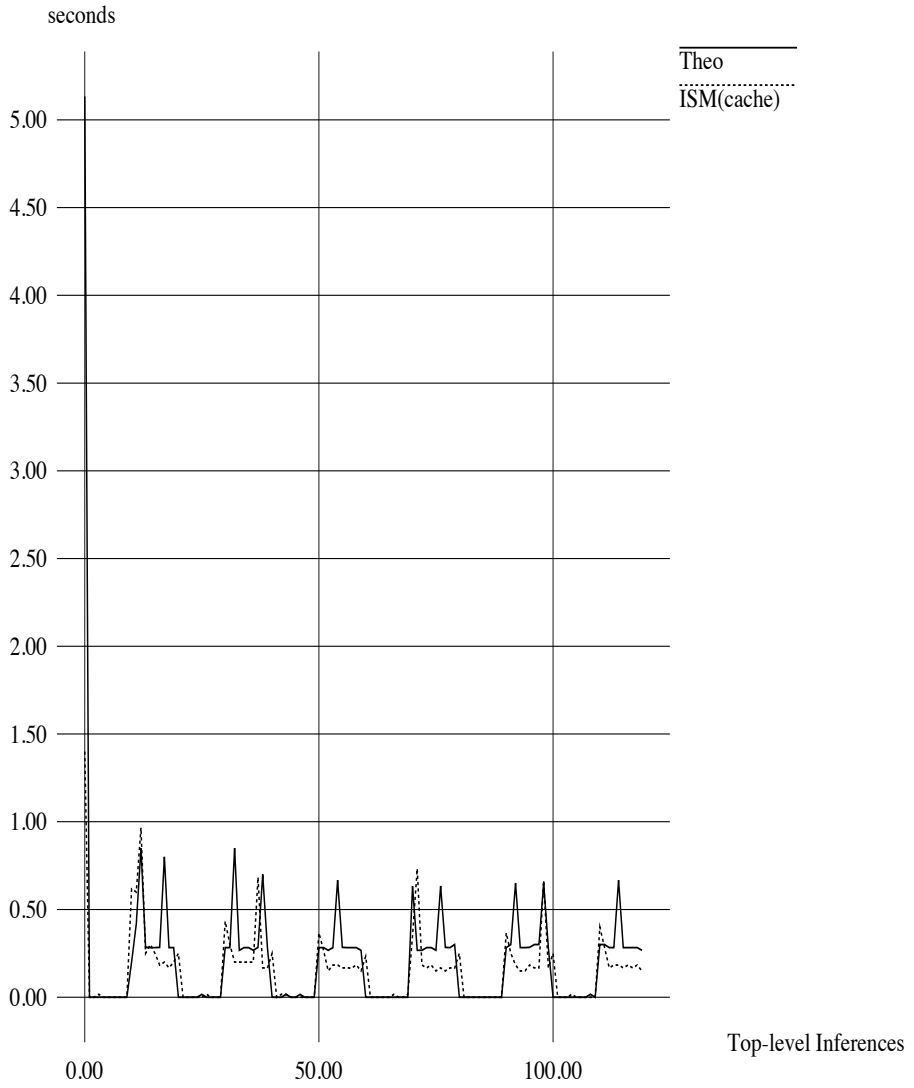

Figure 7.20: Time per Top-Level Inference in an Oscillatory Domain: Theo versus ISM(cache)

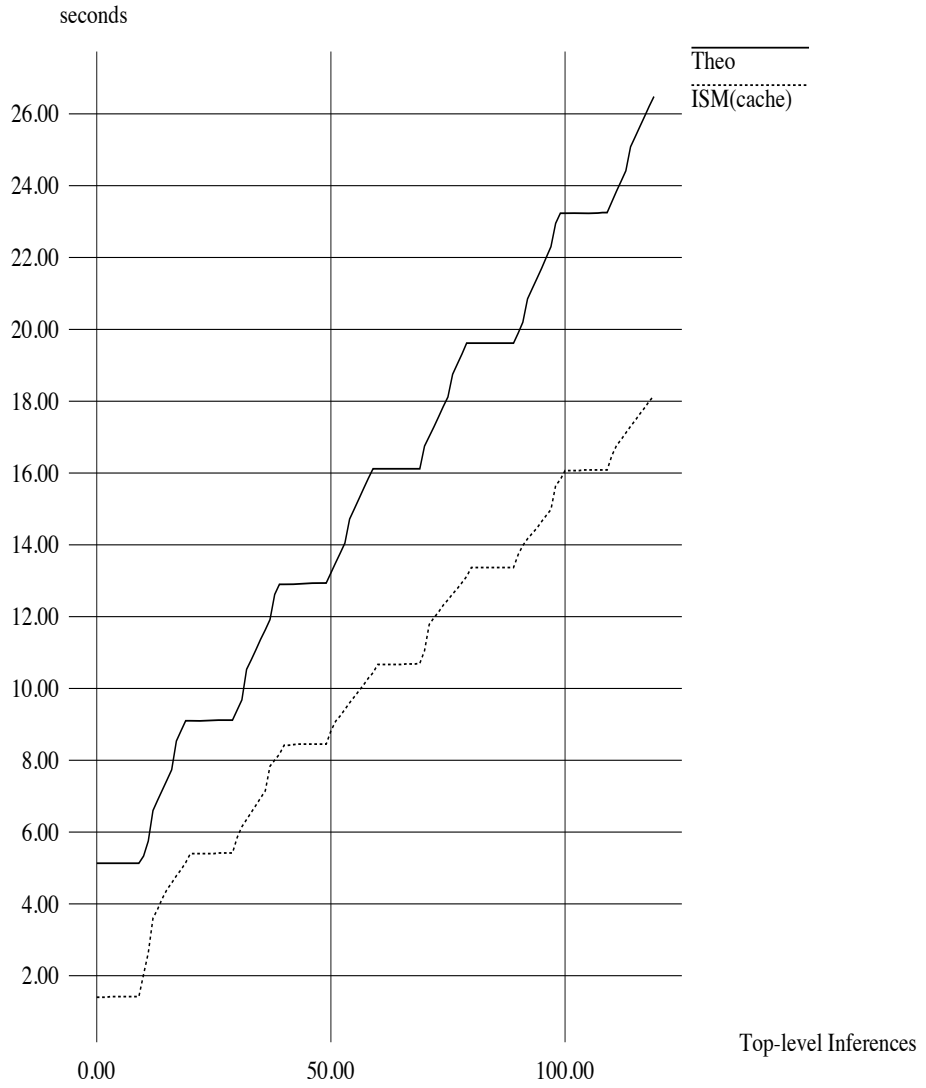

Figure 7.21: Elapsed Time in an Oscillatory Domain: Theo versus ISM(cache)

- ISM's overhead management works well. Sensing overhead is controlled – increasing the number of speedup mechanisms managed by ISM does not significantly decrease architecture performance even when such mechanisms are not useful (i.e., the CAP domain). Furthermore, ISM's sensing control allows it to take advantage of such mechanisms when they are useful, as in the MN domain.

Chapter 8

Related Work

In this chapter, research relating to learning mechanism management and inference mechanism optimization is discussed.

8.1 Learning Mechanism Management

The *utility problem* [Minton 88], in its broadest form, states that that inappropriate learning can degrade the performance of an architecture. Researchers have applied the following techniques to handle this problem:

- Utility analyses
- Expressiveness restrictions to bound the costs associated with learning

Only recently have researchers begun to investigate multiple learning mechanism management. A framework for handling multiple learning mechanisms has emerged:

- Goal-Driven Learning (GDL)

These strategies and their relationship to the scheme that this thesis presents are discussed in this section.

8.1.1 Utility Analyses

The initial response to handling the utility problem was to employ utility analyses. Such analyses are used to ensure that learning will actually increase architecture performance – a speedup mechanism is invoked only

when its utility is positive. Hence, utility analyses provide a means of managing learning mechanisms by causing learning mechanisms to be applied selectively. This strategy has been used successfully and extensively [Etzioni 90] [Gratch 92] [Greiner 92].

In a similar scheme, some systems [Minton 88] compute rough initial utility estimates of the speedup mechanism, then refine these estimates at run-time. This allows the system to determine more accurately such quantities as application-time of an EBG rule, for example.

The most important difference between this approach and that of ISM's is that this approach does not consider *multiple* speedup mechanisms. For instance, [Minton 88] [Etzioni 90] [Gratch 92] [Greiner 92] consider utility for only explanation-based learning. Hence, the problem of managing multiple speedup mechanisms is not confronted. ISM handles the more general problem of multiple mechanism management, which raises the following issues:

- Interactions between speedup mechanisms: Which mechanism or mechanisms should be applied, if more than one is potentially appropriate in a given situation? Do the mechanisms interfere? Are they synergistic?
- Overhead management: managing multiple mechanisms usually requires more sensing and more decision-making, increasing overhead. How can be ensure that this overhead does not swamp the system?

Much of the utility analysis research (e.g., [Gratch 92], [Greiner 92]) assumes the existence of a set of samples that are representative of the entire actual problem distribution. These kinds of analyses attempt to optimize speedup mechanism use. In this approach, the utility analyses are essentially conducted off-line. These analyses are not well-suited to environment dynamics, since any such analysis is *fixed.* Modifying the strategy to account for environment dynamics requires offline retraining. The goals of this thesis are different – ISM does not attempt to optimize speedup mechanism use, it merely seeks to guarantee architecture speedup. For this research, adaptivity – reacting to changes in the environment, such as query distribution shifts and changes in knowledge stability – is considered more important than optimality.

8.1.2 Restricting Expressiveness

One way to guarantee that speedup mechanisms cannot result in decreased architecture efficiency is to ensure that the costs associated with applying

a mechanism – and using its results – are negligible. One example of this technique is the use of unique attributes in chunking [Tambe 91], which reduces the match cost of rule application.

Such techniques can be useful [Brachman 85] [Patel-Schneider 84] [Kacz 86], but can also reduce the effectiveness of the speedup mechanisms. Moreover, the focus of such techniques is very different than the focus of this thesis. Rather than ensuring that speedup mechanisms cannot "hurt" the system, this thesis seeks to use them *appropriately*. Speedup mechanism effectiveness need not be decreased. The "restricting expressiveness" approach handles the cost-benefit tradeoff of speedup mechanisms by minimizing their costs so effectively that the mechanisms can be used all of the time. Unfortunately, this also reduces their benefits. The ISM approach does not try to make speedup mechanisms "universal-applicable." It handles the cost-benefit tradeoff by applying speedup mechanisms only when the benefits outweigh the costs. Furthermore, as in the case of [Tambe 91], restricting expressiveness is used for only single speedup mechanisms.

8.1.3 Goal-Driven Learning

In the Goal-Driven Learning (GDL) [Cox 94] [Gratch 94] [Ram 94] paradigm, the goals of the system are used to make decisions about when and where learning should occur, and which learning strategies are appropriate for a given situation. The effectiveness of GDL depends on the system's ability to make these decisions. GDL systems typically support multiple learning mechanisms; hence, at an abstract level, the goal of GDL is similar to that of this thesis: flexible management of learning mechanisms.

The operation of GDL systems typically consists of three subtasks:

- Blame assignment
- Deciding what to learn
- Strategy selection

In this paradigm, the architecture is given a task. As it performs this task, it maintains a trace reflecting its reasoning process. At a suitable point, the trace is evaluated relative to the architecture's goals. If any failures have occurred, learning is needed to avoid similar problems in the future. Blame assignment determines an explanation of the failure. From this, the system can determine the learning goals which, if achieved, can reduce the likelihood of repeating the failure. Given the learning goals, the architecture

decides which learning algorithm to employ by reasoning about the relative merit of alternative learning strategies in the current situation.

How does the GDL paradigm relate to that of this thesis? ISM makes its decisions based on statistics from many traces. GDL makes choices based on single traces.

Moreover, GDL and ISM conduct learning mechanism management in very different ways. This thesis assumes that architecture environments can be unstable, and that effective learning mechanism management requires *real-time* adaptivity to environment dynamics. Hence, this thesis conducts its management decisions *on-line* – during the performance task. On the other hand, GDL systems assume the ability to *reflect* on system performance, examining past execution traces. This have been done *off-line* to preserve performance. Note that any off-line learning mechanism management strategy cannot, in general, adapt in real-time to environment dynamics. Such systems require time between environment changes to determine new management strategies. Similarly, some systems determine learning strategies based on fixed domain characteristics, such as fixed query distributions, and also cannot adapt to environment dynamics in real time. Examples of systems with this characteristic include [Cox 94] [Ram 94] [Gratch 94].

Furthermore, on-line learning mechanism management necessitates the confrontation of issues such as overhead management and sensing control, to ensure that the execution cost of the management mechanism itself is small. GDL does not address these issues.

Although GDL systems typically manage multiple learning mechanisms, in general, they do not support "competing" learning mechanisms. That is, for a given situation, at most one learning mechanism can be applied – the mechanisms apply to different situations. Consequently, choosing between algorithms is straightforward. For example, in Michalski's Multistrategy Task-Adaptive Learning framework [Michalski 91], learning mechanism management is handled as follows: the relationship between input given to the system and the system's background knowledge specifies what learning algorithm to use. More precisely,

- If the input is not entailed by the system's background knowledge, constructive induction is applied.
- If the input is implied by, or implies a part of the background knowledge, analytic learning is applied.
- If there is a high-level similarity between the input and the background

knowledge, analogical learning is applied.

Because Michalski's framework utilizes learning mechanisms with distinct "domains of applicability," the management task is relatively trivial. Systems sharing this characteristic can be found in [Cox 92] [Cox 94] [Kocabas 94] [Earl 94] [Cox 91] [Spears 91] [Holder 91].

In contrast, this thesis considers the more difficult problem of managing "competing" learning mechanisms. ISM manages three mechanisms, all of which might be applicable to a situation, making the management task much more complex.

8.2 "Static" Inference Mechanism Optimization

Theo's behavior is programmed, or specified, via its knowledge base. ISM's static inference mechanism optimization scheme analyzes this "program," constructing an inference boundary. Any inference outside this boundary at guaranteed fail. ISM prunes such inferences, increasing efficiency.

In some sense, this scheme is a form of compilation. Essentially, this algorithm "compiles" the architecture. That is, the algorithm takes as input a description of the architecture's operation, and restructures the system to operate more efficiently.

Perhaps because learning architectures have developed relatively recently, there has been virtually no work done in the "architecture compilation" area. However, because ISM's algorithm operates only if the system behavior specification is declarative, the algorithm could be considered a form of knowledge compilation. In particular, ISM's algorithm can be considered a form of *partial evaluation*; it produces a "specialized architecture" by incorporating knowledge about restrictions on system inputs. That is, through ISM's knowledge base analysis, ISM "knows" the inputs – knowledge base values – relevant to the system's behavior, and assumes that these inputs are stable. These values determine the inference optimizations that ISM can perform.

Although ISM's algorithm is similar to forms of compilation, there are differences. Compilation typically translates a declarative specification into a more efficient, procedural form [Keller 91]. ISM's algorithm, however, does not generate a non-declarative procedure. Instead, it generates another declarative specification that guides the pruning of Theo's inference process. Because this description is declarative, it is modifiable at run-time,

and such modifications do not require any recompilation. That is, in a typical compilation process, any "code" modification – in this case, architecture specification modification – requires recompilation to incorporate the modifications into the system's operation. In contrast, ISM's algorithm generates a *high-level specification* rather than a *low-level procedure*. Theo *interprets* this specification to prune search. Hence, to incorporate any modifications to the architecture behavior specification, it is only necessary to modify the ISM-generated specifications. Since these specifications are interpreted by Theo, Theo's operation reflects these modifications immediately, without recompilation. Consequently, ISM's algorithm provides more flexibility than forms of compilation.

ISM currently does not take advantage of this flexibility – it assumes the stability of certain knowledge base values – values specifying architecture operation – and if these values are modified, the algorithm must be reinvoked. However, this assumption can be relaxed. Sensors can be embedded into the system monitoring for modifications to the architecture specification values. If any of these values change, the specifications guiding the inference pruning process can be modified appropriately.

Etzioni's STATIC [Etzioni 90] system has some similarities to ISM's load-time optimization technique. As its name implies, STATIC increases architecture efficiency using a static analysis of the architecture's domain theory. EBL [Minton 88] is applied on rules which have a particular characteristic: they must be non-recursive. Hence, STATIC works by analyzing the form of the *domain theory*.

On the other hand, ISM exploits the inference search boundary defined via its knowledge base analysis. This boundary is not delimited by the form of the domain theory. Rather, it is dependent on the *locations* where relevant data initially resides. In effect, ISM relies on the structure of the *knowledge base* to optimize inference.

8.3 Summary

The following is a summary of the characteristics of this research that distinguish it from other research in the field.

- Manages multiple, "competing" speedup mechanisms.
- Computes utilities "on-line," ensuring adaptivity.
- Explicitly confronts overhead issues.

- Trades off architecture flexibility for efficiency where possible by analyzing the structure of the knowledge-base to prune unnecessary inference.

Chapter 9

Conclusion

This thesis investigated ways in which system efficiency can be increased. Two strategies for increasing architecture performance were examined:

- Architecture-controlled management of speedup mechanisms
- Increasing system inference efficiency via control knowledge derived from "static" knowledge base analyses

The system that explores these issues – ISM – uses the Theo learning architecture as an experimental testbed.

This chapter summarizes the important issues and results from the thesis, and presents future areas of research.

9.1 Automatic Speedup Mechanism Management

To manage speedup mechanisms effectively, the architecture must be able to determine *which, when*, and *where* the different mechanisms should be applied. This problem is made more complex by the possibility that the architecture's environment can change, and any competent management strategy must be able to respond appropriately and quickly to such changes. How can this be done? ISM takes the following approach: an agent is embedded into the architecture, and continually observes its operation. The agent is responsible for managing the architecture's speedup mechanisms. For such an agent to operate successfully, the following issues must be resolved:

- How does the agent make its decisions?

- How can the agent overhead costs be minimized?

These issues are examined in turn.

9.1.1 Decision Criteria

ISM bases its management strategy on two elements:

- Speedup mechanism utility analysis
- Application criteria

Utility Analysis

ISM must know how useful a speedup mechanism is in a given situation for it to make effective management choices. Speedup mechanism utility analyses give ISM that information. To generate utility analyses, this thesis uses the following strategy:

- Determine the characteristics that affect the utility of each of the speedup mechanisms
- Determine each mechanism's *ideal utility* – utility given perfect and complete knowledge about the architecture's operation
- Design implementable sensors that can reasonably *approximate* the information needed by the ideal utility formulas
- Modify the ideal utility formulas to use "real" data generated from the sensors instead of the unavailable perfect/complete data

With this strategy, the accuracy of ISM's utility estimates can be easily judged – the strengths and weaknesses of ISM's estimates are simple to determine. Because this analysis makes explicit the effects of sensor design compromises on utility calculations, the task of making tradeoffs – such as sensor efficiency versus accuracy – is also simplified.

Using Utility Estimates

ISM's utility formulas for a mechanism are calculated *independent* of the other mechanisms. Consequently, it is unclear how to use these estimates. Using an *aggressive* strategy, ISM would invoke all mechanisms whose utilities were positive. With a *conservative* approach, ISM would invoke the

single speedup mechanism with the largest estimated utility. Both these approaches have problems, however. The former can lead to interactions which decrease system efficiency, while the latter is overly restrictive.

ISM uses a strategy that allows it to simultaneously invoke mechanisms that *cannot* interact in a negative manner. If there are potential interaction problems, ISM invokes only the mechanism with the largest utility. Hence, this strategy combines the advantages of the aggressive and conservative approaches.

9.1.2 Overhead Management

Assuming that ISM's management decision criteria are sound, one would normally expect architecture efficiency to maximize as the percentage of query instances for which ISM manages speedup mechanisms approaches 100%. However, this is not the case. This speedup mechanism management strategy is expensive; sensing and utility calculation overhead can actually decrease system performance, even with good management decisions. To manage overhead, ISM uses two techniques: *adaptive* sensing and *phasic* sensing.

Adaptive Sensing

A thorough examination of Theo's inference mechanism reveals that the speedup mechanisms handled in this thesis are useful only under certain conditions. If sensing and utility calculations were limited to query instances that satisfy these conditions, a significant amount of overhead could be reduced. This is the basis of ISM's *adaptive sensing strategy*. However, because the query instances for which these conditions are true continually change, a second set of sensors are needed to determine when ISM's sensors need to be applied. Hence, this strategy involves a sensing trade-off. ISM's expensive utility calculation sensing is reduced, but other sensing is increased.

Phasic Sensing

Phasic sensing is based on the observation that architecture operation tends to *stabilize* after some time. That is, after an initial "transient" period, ISM management strategy needs to be modified only rarely. Note that this is not necessarily true at the user-defined ground level. However, it is true at the system-level. Because the system level tends to dominate the ground level,

in general, this assertion is true. Hence, phasic sensing control consists of two phases:

- Decision phase: phase during which ISM operates normally
- Error-detection phase: ISM stops managing speedup mechanisms, relying on previous management decisions. This can drastically reduce sensing costs. To ensure adaptivity, however, ISM monitors for "serious" management decision errors. If enough such errors are detected, ISM transitions back to the decision phase.

Because sensing for serious management errors is much cheaper than full sensing or even adaptive sensing, this strategy is very effective at reducing ISM overhead.

9.2 "Static" Inference Mechanism Optimization

Any inference engine's inference mechanism consists of a search through the knowledge base to find data relevant to the query instance. In some systems, such as Theo, this search can be very inefficient, exploring many unsuccessful paths. Theo's inefficiency results directly from its uniformity and flexibility. Because Theo's behavior is not "hard-coded," Theo must initiate many system-level inferences to determine its operation. Because of Theo's uniformity, these inferences do not terminate immediately – they result in other inferences.

Flexibility and uniformity in a learning architecture can be very valuable. However, as Theo demonstrates, such traits have a high price – poor efficiency. ISM uses a technique to increase efficiency without sacrificing flexibility and uniformity. By examining the system knowledge-base prior to run-time, ISM determines the situations under which various search paths *cannot* succeed. The inference *methods* of sets of query instances and the locations of relevant query instance values reveal this information. At run-time ISM uses this data to prune Theo's inference search paths, increasing efficiency. This scheme is able to *adapt* to the requirements of the domain. Domains requiring flexibility will not be hampered. Domains not requiring flexibility will execute more efficiently.

9.3 Performance

ISM has been tested in two domains with widely varying characteristics:

CAP: Stable knowledge base, uniform query distribution, widely varying query instance inference structures, complicated inferences, high percentage of novel system-level inferences

MN: Unstable knowledge base, spiky query distribution, similar query instance inference structures, simple inferences, very few novel system-level inferences

In the CAP domain, Theo with ISM outperformed Theo by about a factor of 2. In the MN domain, after an initial "training period," Theo with ISM outperforms Theo by more than a factor of 12. ISM's effectiveness is not specific to domains with very particular characteristics; it appears to be generally useful.

ISM has also been tested in a domain with oscillatory characteristics. Its performance in this domain shows that ISM is adept at responding to environment changes.

9.4 General Lessons

Although ISM has been implemented only in Theo, and hence has many Theo-specific characteristics, this purpose of this thesis was to investigate the broader, general issues involved with automatic speedup mechanism management and inference mechanism optimization techniques. This section summarizes the features for which ISM is most useful, and identifies system and domain characteristics upon which ISM relies.

9.4.1 Automatic Speedup Mechanism Management

Architectural Features

The following architectural features are needed to implement an ISM-like embedded agent system:

- *Ability to embed sensors and effectors into the architecture.* Without this ability, an embedded agent cannot observe architecture operation and cannot invoke operations to increase system efficiency.

- *Ability to monitor relevant data.* For an agent to make reasonable decisions, it must have reasonably accurate data from which to base its decisions. Some of ISM's sensors are built into Theo – such as Theo's explanation facility. Other sensors have been fabricated.

- *Ability for agent to operate efficiently.* As ISM's experiments show, it is imperative that agent overhead be low. Some form of sensing control may be needed, such as ISM's adaptive/phasic sensing. A more radical solution would be to parallelize the agent and architecture operation. Efficient agent operation usually implies that it is necessary to be able to implement simple, inexpensive, accurate sensors. Likewise, utility calculations must be fast. Furthermore, if sensing control is needed, the architecture operation must be such that effective sensing control is possible. For instance, adaptive sensing is very effective for ISM because of Theo's high percentage of meta-level inference. For another architecture, that strategy may not be useful.

The following architectural characteristics increase the effectiveness of an ISM-like system:

- *One or more speedup mechanisms.* The larger the set of available speedup mechanisms, the more useful an ISM-like agent is. Numerous speedup mechanisms can become very difficult to manage well via any other technique, due to the large number of options available, and possible interactions between mechanisms. However, an ISM-like agent could very well be useful for even a single speedup mechanism, if it is prone to utility problems.

- *Simple default speedup mechanism management approach.* It is unlikely that any simple management approach, such as Theo's, makes good use of the available mechanisms. Hence, it becomes more likely that a more sophisticated strategy, such as ISM, would increase system performance.

- *Complicated operation.* Complicated systems are difficult for anyone to tune. It becomes impossible to grasp all the information necessary to optimize speedup mechanism usage, especially for multiple domains with differing characteristics. In such cases, automated management is the only way to tune speedup mechanisms at a fine grain.

Speedup Mechanism Features

Speedup mechanism characteristics can have a dramatic effect on the efficacy of an ISM-like system. Some speedup mechanism features that increase the usefulness of ISM are given below.

- *Mechanism has high cost if used inappropriately.* Theo's caching, EBG, and BEBG have this feature. Without any cost, mechanisms can always be applied without degrading system performance.

- *Mechanism has high-risk/high-payoff characteristic.* Any fixed management strategy is not likely to utilize such potentially expensive mechanisms. The chances of such mechanisms "backfiring" is too large. However, because an ISM-like agent can analyze risk/payoff tradeoffs well, it can manage such mechanisms effectively, making use of their high payoffs.

- *Available mechanisms are useful in different situations.* An ISM-like agent has the opportunity to combine the strengths of all the algorithms. If they are strong in the same situations, the ability to choose between mechanisms is not useful.

- *Inexpensive to estimate mechanism utility.* If utility calculations are too time-consuming, management overhead will overwhelm any efficiency gains.

Domain Features

Some domain features that make automatic speedup mechanism management particularly effective are:

- *Dynamic environment.* An ISM-like agent can adapt to environment dynamics, resulting in performance superior to any fixed management scheme.

- *Quasi-stable environment.* It is helpful if domain dynamics are not too abrupt. This allows sensors to track the changing factors necessary in determining speedup mechanism utility.

- *Complicated operation.* Complex domains are difficult for humans to tune.

- *Domain not well suited to default speedup mechanism management strategy.* If the default architecture management strategy is fixed, the characteristics of some domains may cause the applied mechanisms to actually reduce system efficiency. For example, in a very dynamic domain, caching all query instances (by default) may slow the system down.

9.4.2 Static Inference Optimization

Architectural Features

ISM's static inference optimization technique is most useful for architectures with the following characteristics:

- *Uniformity and flexibility.* These features tend to cause the architecture to explore many potentially unsuccessful search paths.
- *No mechanism that decreases amount of unsuccessful search over time.* These mechanisms "compete" with the static optimization technique, limiting its usefulness. Caching is potentially such a competing mechanism – when queries are repeated and knowledge base values are stable, caching reduces search very effectively, obviating the need for static optimization.
- *Declaratively represented inference methods.* Such a characteristic would allow broader applicability of the optimization technique.

Domain Features

ISM's static inference optimization technique is most useful for domains with the following characteristics:

- *High percentage of novel inference.* Novel inferences tend to result in more knowledge-base search, increasing the likelihood that the inference optimization technique would be useful.
- *No modification of optimized query instance values.* In domains without this feature, any static analysis becomes invalid, rendering the optimization technique useless.

9.5 Future Work

Automatic Speedup Mechanism Management

- *Augment ISM to handle additional speedup mechanisms* – ISM currently handles only a small set of the existing speedup mechanisms. Moreover, the mechanisms handled by ISM are very similar in a sense. Handling radically different mechanisms might give ISM much more speedup leverage. Understanding the effects of speedup mechanism characteristics on an ISM-like agent could be very interesting.

- *Apply learning to ISM* – In some sense, ISM's power is very limited because it tends to make very small-scale management decisions. The ability to generalize well – using learning mechanisms – could radically alter ISM's performance.

- *Consider user-architecture interactions* – Giving ISM the ability to interact with the user could drastically increase ISM's power. This interaction would allow ISM to loosen its restrictions, and base its goals on the needs of the user, giving it latitude to try many additional kinds of speedup strategies.

"Static" Inference Optimization

- *Automated handling of methods* – ISM currently handles only a few methods. It would be useful to investigate the means to give ISM the capability to analyze methods autonomously, given, say, declarative descriptions of methods. This would dramatically increase the generality – and effectiveness – of this technique.

Bibliography

[Brachman 85] Brachman, R., Gilbert, P., and Levesque, H.
An Essential Hybrid Reasoning System: Knowledge and Symbol Level Account of Krypton.
In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 532-539. 1985.

[Cox 91] Cox, Michael and Ashwin, Ram
Using Introspective Reasoning to Select Learning Strategies
In *Proceedings of the First International Workshop on Multistrategy Learning*, pages 217-230, Center for Artificial Intelligence, George Mason University, 1991

[Cox 92] Cox, Michael and Ram, Ashwin
Multistrategy Learning with Introspective Meta-Explanations
In *Machine Learning: Proceedings of the International Workshop*, pages 123-128. ML92, Morgan-Kaufman, 1992

[Cox 94] Cox, Michael and Ram, Ashwin
Choosing Learning Strategies to Achieve Learning Goals
In *AAAI-94 Spring Symposium Series: Goal-Driven Learning*, pages 12-21

[Earl 94] Earl, Charles and Firby, James
An Integrated Action and Learning System
In *AAAI-94 Spring Symposium Series: Goal-Driven Learning*, pages 22-27

[Etzioni 90] Etzioni, Oren
A Structural Theory of Search Control.
Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 1990.

[Gratch 92] Gratch, Jonathan and DeJong, Gerald
COMPOSER: A Probabilistic Solution to the Utility Problem in Speed-Up Learning
In *Proceedings from the Tenth National Conference on Aritificial Intelligence*, pages 235-240. AAAI-92, MIT Press, 1992

[Gratch 94] Gratch, Jonathan, DeJong, Gerald, and Chien, Steve
Deciding When and How to Learn
In *AAAI-94 Spring Symposium Series: Goal-Driven Learning*, pages 36-45

[Greiner 92] Greiner, Russell and Jurisica, Igor
A Statistical Approach to Solving the EBG Utility Problem.
In *Proceedings from the Tenth National Conference on Aritificial Intelligence*, pages 241-248. AAAI-92, MIT Press, 1992

[Holder 91] Holder, Lawrence
Selection of Learning Methods Using an Adaptive Model of Knowledge Utility
In *Proceedings of the First International Workshop on Multistrategy Learning*, pages 247 -256, Center for Artificial Intelligence, George Mason University, 1991

[Kacz 86] Kaczmarek, T., Bates, R., and Robbins, G.
Recent Developments in NIKL.
In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 978-985. 1986.

[Keller 91] Keller, Richard
Applying Knowledge Compilation Techniques to Model-Based Reasoning.
IEEE Expert, 6(2):82-87, 1991

[Kocabas 94] Kocabas, Sakir
Goal Directed Discovery and Explanation in Particle Physics
In *AAAI-94 Spring Symposium Series: Goal-Driven Learning*, pages 54-61

[Laird 87] Laird, J., Newell,A., and Rosenbloom, P.
SOAR: An Architecture for General Intelligence.
Artificial Intelligence 33(1):1-64, September, 1987.

[Michalski 91] Michalski, R.
Inferential Learning Theory as a Basis for Multistrategy Task-Adaptive Learning
In *Proceedings of the First International Workshop on Multistrategy Learning*, pages 3-18, Center for Artificial Intelligence, George Mason University, 1991

[Minton 87] Minton, S., Carbonell, J., Etzioni, O., Knoblock, Ca., Kuokka, D
Acquiring Effective Search Control Rules: Explanation-Based Learning in the PRODIGY System.
In Langley, P. (editors), *Proceedings of the Fourth Internatinoal Workshop on Machine Learning.* Morgan-Kaufmann, Irvine, June, 1987.

[Minton 88] Minton, Steve
Learning Effective Search Control Knolwedge:
An Explanation-Based Approach.
Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 1988.

[Mitchell 86] Mitchell, T., Keller, R., and Kedar-Cabelli, S.
Explanation-Based Generalization: A Unifying View.
Machine Learning 1(1), 1986

[Mitchell 91] Mitchell, T., et al
Theo: A Framework for Self-Improving Systems
In *Architectures for Intelligence*, K. VanLehn, Ed., Erlbaum, 1991

[Mitchell 93] Mitchell, T., et al
TheoGT
November, 1993
Internal Theo Project Working Paper.

[Mitchell 94] Mitchell, T., et al
Experience With A Learning Personal Assistant.
In *Communications of the ACM*, 1994.

[Patel-Schneider 84] Patel-Schneider, P.F.
Selective Backtracking.

In Clark, K. L. and Tarnlund, S. A. (editor), *Logic Programming*. Academic Press, New york, New York, 1982.

[Ram 94] Ram, Ashwin and Leake, David
A Framework for Goal-Driven Learning
In *AAAI-94 Spring Symposium Series: Goal-Driven Learning*, pages 1-11

[Spears 91] Spears, William and Gordon, Diana
Adaptive Strategy Selection for Concept Learning
In *Proceedings of the First International Workshop on Multistrategy Learning*, pages 231-246, Center for Artificial Intelligence, George Mason University, 1991

[Tambe 91] Tambe, Milind
Eliminating Combinatorics from Production Match
Ph.D. thesis, Computer Science Department, Carnegie Mellon University, 1991.