Performance Aspects of GBB

Kevin Q. Gallagher
Daniel D. Corkill

April 1989
COINS Technical Report 89-54

To appear in *Blackboard Architectures and Applications*, V. Jagannathan, Raj T. Dodhiawala, and Larry Baum, editors, Academic Press, 1989.

Performance Aspects of GBB

Kevin Q. Gallagher and Daniel D. Corkill

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

April 1989

Abstract

The run-time performance of a blackboard-based application can be significantly improved by selecting an appropriate blackboard database representation. When numerous objects reside on the blackboard, retrieval costs can be signifcantly reduced by limiting search to the blackboard area containing the desired objects. In this paper, we detail the blackboard representation and retrieval algorithms used by GBB. GBB allows the blackboard representation to be easily modified to optimize the performance of blackboard operations. We present empirical results demonstrating the effect of tuning the blackboard database in a large application. The results underscore the importance of efficient blackboard database operations and the benefits of a flexible, instrumented blackboard development environment. We conclude with some general guidelines for tuning the representation to produce the best performance.

1 Introduction

The performance of blackboard-based applications can be significantly enhanced by an appropriate blackboard database implementation. The blackboard paradigm relies heavily on the blackboard for knowledge source (KS) interaction and for holding tentative, partial results until they are needed. Although published measures are non-existent,[1] the amount of processing time devoted to blackboard interaction is significant—even in applications built with blackboard database machinery that has been customized for speed. Therefore, the runtime performance of a blackboard-based application is strongly influenced by the efficiency of placing and retrieving blackboard objects.

The blackboard acts as an associative memory for KSs. Objects are held on the blackboard to be used when and if they are needed by the KSs. When numerous objects reside

This research was sponsored in part by donations from Texas Instruments, Inc., by the Office of Naval Research, under a University Research Initiative Grant (Contract N00014-86-K-0764), and by the National Science Foundation under CER Grant DCR-8500332

[1] An exception is Fennell and Lesser's measurements with an early version of the Hearsay-II speech understanding system which showed a blackboard interaction to KS processing ratio of 10/17 [2]. The blackboard-interaction/processing ratios of the Distributed Vehicle Monitoring Testbed (used in these experiments) range from 8/19 15/3, depending on how efficiently the blackboard is implemented.

on the blackboard, retrieval costs can be signifcantly reduced by limiting search to the blackboard area containing the desired objects. Partioning the blackboard into levels is one means of reducing search. The Generic Blackboard Development System (GBB) [3,4] further reduces search by representing each level as a highly-structured, n-dimensional volume. Blackboard objects occupy an extent within the n-dimensional volume based on the values of *dimensional index* attributes. In this paper, we detail the blackboard representation and retrieval algorithms used by GBB; we present empirical results demonstrating the performance improvements that were obtained by tuning the blackboard database in a large application: the Distributed Vehicle Monitoring Testbed [5,6]; and we present general guidelines (based upon blackboard density and blackboard insertion/retrieval ratios) for tuning the representation to produce the best performance.

2 GBB Blackboard and Unit Representation

GBB represents the blackboard database as a hierarchical tree[2] of nested *blackboards*, the leaves of which are primitive blackboard elements called *spaces*. For example, spaces would be used to implement the problem solving levels in the Hearsay-II speech understanding system [7]. GBB views each space as a structured n-dimensional volume. This space dimensionality provides a *metric* for positioning blackboard objects, called *units*, onto the blackboard in terms that are natural to the application domain. A space dimension can be either *ordered*, real numbers in some interval; or *enumerated*, members of a set of labels.

Units occupy some n-dimensional extent within the space's dimensionality. A unit is located on a space based on the values of its *dimensional indexes* (often abbreviated to just *indexes*). Each index corresponds to a space dimension.[3] Each unit defines how its index values will be computed from its slots (attributes). For example, a single slot may contain both the x and y dimensional indexes for a unit.

There are two types of indexes, *scalar* indexes and *composite* indexes. A scalar index represents a single "atomic" value. For ordered dimensions the scalar index types are *points* (a single number) and *ranges* (two numbers representing minimum and maximum values). For enumerated dimensions an atomic index value is a single label.

A composite index represents a group of atomic index values. Composite indexes are divided into two types, *sets* and *series*. A set index is an unordered group of dimensional index values. A series index is a group of dimensional index values where the order of the elements is determined by the value of another dimensional index. For example, suppose an application is monitoring a chemical reaction. A series of temperature observations over time would be implemented as a series index. The two dimensional indexes in this example are *time* and *temperature*. *Time* is called the *ordering index* — it is the index by which the *temperature* observations are ordered. Several non-ordering indexes may be ordered by the same ordering index (e.g., a series of temperature and pressure observations over time), however the current implementation does not support more than one composite index.

[2]Actually, it can be a forest.

[3]The set of indexes need not exhaust the set of space dimensions. However, such units can not be retrieved using the missing dimensions.

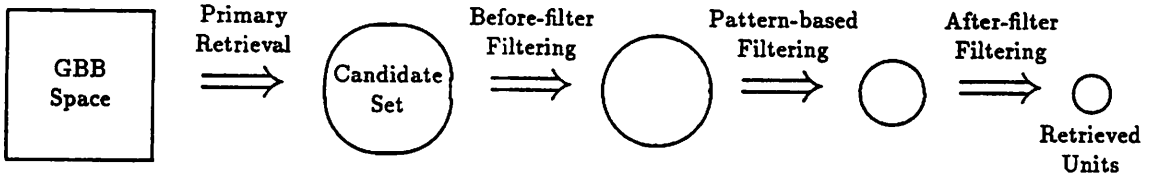


Figure 1: GBB's Unit Retrieval Steps.

It is important to note that the distinction between scalar and composite values is not a property of space dimensions; it is a property of indexes. That is, each unit class specifies whether an index is scalar or composite. To summarize:

- Spaces have dimensions.
- Units have indexes.
- An index is a particular value or set of values for a dimension.

In database terms, indexes are *keys*, but in the case of composite indexes, a single key can be quite complex.

3 Blackboard Retrieval

3.1 Overview of Blackboard Retrieval

GBB retrieves units from a space based on a *retrieval pattern*. The pattern specifies an n-dimensional volume of the blackboard in which the desired units will be found. In addition to the pattern, a retrieval request can specify two procedural filters, called the *before-filter* and the *after-filter*. These filters check for conditions that can not be expressed in the pattern. For example, a filter might compute a complex predicate on the unit, or check attributes that are not represented in the dimensionality of the space.

The retrieval process is divided into four steps: *primary retrieval*, *before-filter* filtering, *pattern-based* filtering, and *after-filter* filtering (Figure 1). To simplify the exposition, we will describe each step as if the result of one step is passed on to the next step. In fact, the steps are interleaved to avoid creating temporary lists of intermediate results. In the following description, we will concentrate on the retrieval mechnanisms used for ordered dimensions.

The primary retrieval step selects a set of units that might satisfy the pattern. This is called the *candidate set*. Each member of the candidate set is then passed through the three filtering steps. Units that satisfy all the remaining steps are collected into a list and returned as the result of the retrieval. Because the filtering steps consider only units in the candidate set, the primary retrieval step must select *all* units that could possibly satisfy the remaining steps.

The pattern-based filtering step compares each unit to the pattern to see if the unit satisfies the constraints of the pattern. This step is necessary because membership in the candidate set does not guarantee that a unit will satisfy the pattern. The before-filter and after-filter filtering steps simply apply the filter predicates to each unit. If a predicate

returns false then the unit is removed from further consideration. The goal of efficient retrieval is to efficiently minimize the number of units in the candidate set which do not satisfy the pattern. This will minimize the number of units that must be examined in the remaining steps.

Each space has a *unit mapping* which specifies how the storage of units on that space is implemented. The simplest implementation strategy is to store all a space's units in a list. In this case the primary retrieval step is very simple — the candidate set is simply the entire list of units on the space. However, the filtering steps must be applied to every unit on the space. This is an ineffecient strategy when many units do not satisfy the retrieval constraints.

The problem with the simple list strategy is that the primary retrieval doesn't reduce the candidate set at all. A better strategy is to partition a space's storage into buckets or cells based on its dimensional attributes. (This is similar to the partition and grid range searching strategies [1].) The primary retrieval step can then examine the pattern to determine which buckets should be included in forming the candidate set. These buckets are called *candidate buckets*. The candidate set is the union of the contents of all the candidate buckets.

The bucket technique is appropriate for ordered indexes, which have the property of *neighborhood*. That is, there is a notion of one index value being "near" another index value. Examples of such indexes are physical dimensions such as x and y location and *time*. A hash table can not easily represent this neighborhood relationship among index values.

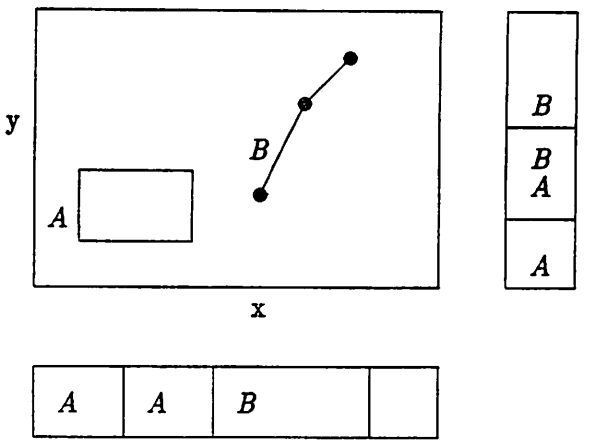
On the other hand, a hash table *is* appropriate for indexes which are unordered such as GBB's enumerated dimension type, which is simply an unordered set of labels.

3.2 Details of Primary Retrieval

As mentioned above, there is a *unit mapping* which specifies exactly how the storage is to be arranged for each unit class that can be stored on space. In particular, the unit mapping specifies what space dimensions will be used in the primary retrieval step. It specifies how to partition these dimensions into separate buckets and the exact arrangement of the data structures.

If a unit mapping specifies that *no* space dimensions are to be used to locate units on a space then the space is implemented as a list. To retrieve units from the space in this case the three filtering steps are applied to every unit on the space because the candidate set is the entire list of units. Units that satisfy the filtering steps are then collected into a list of retrieved units. This representation is desirable only if there are very few units on the space. Typically, however, there will be hundreds or thousands of units on a space.

When one or more space dimensions are specified for the unit mapping then the space is implemented as one or more arrays. Each array dimension represents one space dimension and each array element represents one bucket. Each element contains a list of all the units which fall into that bucket. For example, if a space has the dimensions *time*, x, and y then one possible unit mapping would use three vectors, one for each of the space dimensions. Another representation would use a two-dimensional array for x and y and a vector for *time*. Note that a unit may fall into several buckets in a single array depending on the values of the dimensional indexes for that unit, so the lists of units in each bucket are not



A two dimensional space (x, y) containing two units (A and B). The boxes on the right and the bottom represent two vectors of buckets and show which buckets the two units fall into. Note that the bucket sizes need not be uniform.

Figure 2: Mapping Units to Buckets

disjoint. For example, a dimensional index value may be a range which spans a bucket boundary (Figure 2, unit A), or the elements of a composite index may fall into different buckets (Figure 2, unit B).

A unit mapping may specify that only a subset of the space dimensions will be used in the primary retrieval step. For example, another unit mapping for the three dimensional space above would be two vectors, one for x and one for y.[4]

By examining the pattern, GBB can determine the candidate buckets in each array. This is a straightforward computation based on the values for each dimensional index in the pattern. A minor complication arises when elements of a composite index in the pattern overlap one another. Without special attention, a bucket could be included more than once because separate index elements fell into the same bucket. To avoid unnecessary search, duplicate buckets are removed.

To aid in retrieval processing, each unit has a special internal slot called the *mark* slot which contains a small integer. During the retrieval process, this slot is used to record the status of each unit.

If a space is implemented as a single array then the candidate set is simply the union of the contents of all the candidate buckets. To avoid creating temporary lists each candidate bucket is processed one at a time. Each unit in the bucket is tested by the three filtering steps and marked as having passed or failed. If a unit has already been marked as tested it is not tested again. Those units which pass the filtering steps are collected into a list of retrieved units.

If a space is implemented as more than one array then the candidate set is the intersection of the contents of the candidate buckets for each array. A straightforward intersection

[4]One reason this might be done is that a space may store several different unit classes and all space dimensions may not be useful primary retrieval keys for all classes.

```
;; Clear the mark in the first array's candidate buckets.
(clear-unit-marks (bucket-selection (first space-arrays)))

(let ((count 0))

   ;; Update the mark in the remaining arrays' candidate buckets.
   (dolist (array (rest space-arrays))
      (dolist (unit (bucket-selection array))
         (when (= (get-unit-mark unit) count)
            (incf (get-unit-mark unit))))
      (incf count))

   ;; Collect the units in the intersection.
   (dolist (unit (bucket-selection (first space-arrays)))
      (when (= (get-unit-mark unit) count)
         (filter-and-collect unit))))
```

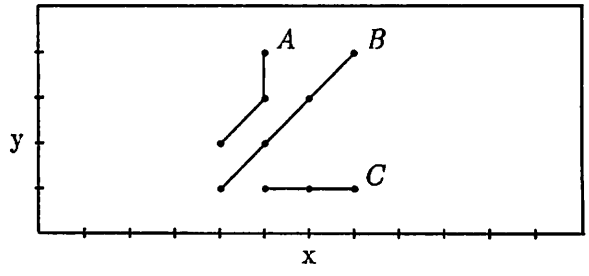
Figure 3: "Tag-based" Intersection

algorithm would require time $O(n^m)$, where n is the number of units on the space and m is the number of separate arrays, as well as space $O(n)$. By using the mark slot associated with each unit we can reduce the time to $O(n)$ and the space to a constant.

This "tag-based" intersection is computed as follows. Select one array as the initial array and set the mark to zero for all units in its candidate buckets. Then consider each unit in the second array's candidate buckets. If the unit's mark is zero then set the mark to one otherwise leave it unchanged. Then consider each unit in the third array's candidate buckets. If the unit's mark equals one then set the mark to two. This continues until we have exhausted all the arrays. Finally, return to the initial array and examine each unit in its candidate buckets. If the unit's mark is equal to n (where n is the number of arrays) then the unit is a member of the candidate set and the three filtering steps are applied. (See Figure 3.) Because the mark was initialized to zero for each unit in the initial array's candidate buckets, only those units that fall into the candidate buckets for all arrays will have the proper value in their mark slots.

As in the single array case above, each unit in the bucket is tested by the three filtering steps and marked as having passed or failed. If a unit has already been tested it is not tested again. Those units which pass the filtering steps are collected into a list of retrieved units.

To illustrate the tradeoffs, consider the three tracks (A, B, and C) depicted below, and suppose the application is looking for tracks which pass through the point (5,3).



If the space is represented simply as a list of units then the primary retrieval step retrieves all three units, which must be compared with the pattern. If the space is represented as two vectors (one for x and one for y), with each bucket one unit wide, then the primary retrieval step selects all units that occupy the $x = 5$ bucket (in this case B & C). This set is intersected with the set of units occupying the $y = 3$ bucket (A & B), for a primary retrieval result set (B). Pattern-based filtering is then applied to each element of this result set.

3.3 Details of Pattern-Based Filtering

The pattern-based filtering step compares, in detail, each unit with the pattern to determine if the unit satisfies the constraints of the pattern. The pattern primarily specifies an n-dimensional blackboard volume (or set/series of volumes) in which the desired units will be found. The constraints imposed by each dimension are tested independently. If a unit fails to satisfy the constraints of any dimension, the unit is eliminated from further consideration. The pattern specifies an *element match* criterion which determines how units are compared to the pattern. The four element match criteria are:

Exact The unit must exactly match the pattern.

Includes The unit must entirely include the pattern.

Within The unit must be entirely within the pattern.

Overlaps The unit must overlap with the pattern.

The distinctions between the different match criteria are only meaningful when one or both of the index elements from the pattern and the unit are ranges. If both the pattern's and the unit's index elements are points or labels then these match criteria are all equivalent.

Several options are available to control the comparison of composites. The *match* and *mismatch* options apply to both set and series indexes. The remaining options only apply to series indexes.

Match This determines how many index elements in the pattern must be matched by index elements in the unit.

Mismatch This is the upper limit on the number of index elements in the pattern that may be mismatched with index elements in the unit.

Skipped This is the upper limit on the number of index elements in the pattern that may be skipped in the unit.

Before and After Extras These options determine how many index elements may appear in the unit either before or after the composite index in the pattern.[5]

Contiguous This determines whether the composite index elements that match must be contiguous or whether mismatching elements may appear between the matching elements.

[5]For example, suppose the pattern has a series of *temperature* observations starting at *time* = 6 and the unit has a series starting at *time* = 4. This unit would pass only if the the before extras option allowed two or more before extras.

Note that, the index element to index element comparisons are still governed by the element match criterion specified in the pattern.

As with unit indexes, indexes in the pattern can be of type scalar, set, or series. So, the following six combinations of unit index and pattern index can occur. The order here is not important. (Comparing a scalar unit index with a set pattern index is treated identically to comparing a set unit index with a scalar pattern index.)

Scalar/Scalar	Set/Set
Scalar/Set	Set/Series
Scalar/Series	Series/Series

To avoid subtle conceptual errors, Scalar/Series comparisons are not allowed.[6] We consider the comparison rules for the remaining five cases below.

Scalar/Scalar In the simplest case, when an index is a scalar in both the pattern and the unit, GBB simply compares the value of the index in the unit with the value of the index in the pattern to see if they match with respect to the match criterion from the pattern.

Scalar/Set The comparison of a scalar index with a set index reduces to a set membership test. This is the case when the index is a scalar in the unit and a set in the pattern or vice versa. If the scalar index is a member of the set index then the comparison succeeds. Each index element from the set is compared with the scalar index element in the same way as for scalar/scalar comparison above.

Set/Set and **Set/Series** For both these cases the comparison is treated as a set intersection. The size of the intersection set must be greater than the number of matches required by the pattern. In addition the number of elements from the pattern that are not in the intersection set must be less than or equal to the *mismatch* criterion from the pattern.

Series/Series In the series/series case the index elements for *corresponding* values of the composite index are compared. Continuing the *time/temperature/pressure* example from above, the values for *temperature* and *pressure* at *time* = 6 in the pattern would be compared with the corresponding values for *temperature* and *pressure* at *time* = 6 in the unit. This continues for each element in the pattern. The unit will pass the pattern-based filtering step if the match, mismatch, skipped, before extras, and after extras criteria are satisfied.

Several patterns can be conjoined. The effect of this is that a unit must satisfy all the patterns to pass the unit/pattern comparison. For conjoined patterns the primary retrieval is based on all the component patterns. Then, during the pattern-based filtering step, GBB will compare each unit to the simplest component patterns first in the expectation that this will "cheaply" eliminate some units early.

[6] For example, in comparing a scalar pattern index with a series unit index, the composite index of the series may not be present in the scalar. It is not immediately apparent whether this situation should be treated as an error. However, the restriction causes the user to be explicit about what is intended. Transforming one type of index to another is a simple operation, so this is not a great limitation.

3.4 Comparison with Other Representations

The partition and grid techniques used in GBB work best when the units are located so that the buckets are uniformly loaded. Two other representation strategies that have been proposed for blackboard database systems are the *k*-d tree [1] and the quadtree [8]. They are appealing because they readily adapt to the data, providing more selectivity where there is more data, and give good performance when little is known, *a priori*, about the distribution of data or queries.

Both quadtrees and *k*-d trees are hierarchical data structures which recursively divide space. There are two general types of quadtrees: *region quadtrees* and *point quadtrees*. The region quadtree is inappropriate for a blackboard database because the entire quadtree represents just one object. The point quadtree is a multidimensional generalization of a binary search tree. Each node in the tree stores one datapoint and has 2^n children. Each child represents one 'quadrent' of the space represented by its parent.

The *k*-d tree is also a generalization of a binary search tree. At each level of the tree a different dimension is tested when determining which branch to take. For example, in the two dimensional case the x dimension is tested at even levels in the tree and the y dimension is tested at odd levels. The *adaptive* *k*-d tree is a refinement of the *k*-d tree where, instead of cycling through the dimensions in a fixed order, choose the dimension that best divides the datapoints in the best way for each node.

While the quadtree and *k*-d tree may use less space than the grid method, there is no speed advantage. The grid method requires $O(R)$ steps on average (where R is the number of objects found), but quadtrees and *k*-d trees require $O(R + \log N)$. In addition, deletion and balancing are quite complicated and time consuming operations on the quadtree and *k*-d tree.

4 An Example of Performance Tuning

In this section we present empirical results demonstrating the the effect of tuning the blackboard database in a large application: the Distributed Vehicle Monitoring Testbed (DVMT) [5,6]. These results are interesting because they were obtained without changing *any* of the problem solving or control activities of the DVMT. Each set of experiments executed the same sequence of KSs, created and retrieved the same blackboard objects, and generated the same solution. The *only* difference between each experiment was the processing time required to insert and retrieve blackboard objects.[7]

Because the DVMT is implemented in GBB, the database implementation can be easily changed without recoding (or even recompiling). Such flexibility is important for two reasons. First, the application writer may not initially understand the insertion/retrieval characteristics of the application; so the representation of blackboard objects is subject to change as design intuition evolves into application experience. Second, the insertion/retrieval characteristics may change from those of the prototype as the application

[7]Several systems such as Joshua [9], MRS [10], and KEE [11] provide abstraction mechanisms for modifying the representation of data structures without changing rules. However, performance improvements using Joshua often involve reductions in the number and changes in the sequence of rule firings; MRS is tailored to logic programming; and KEE leaves you to write your own procedural storage and retrieval functions.

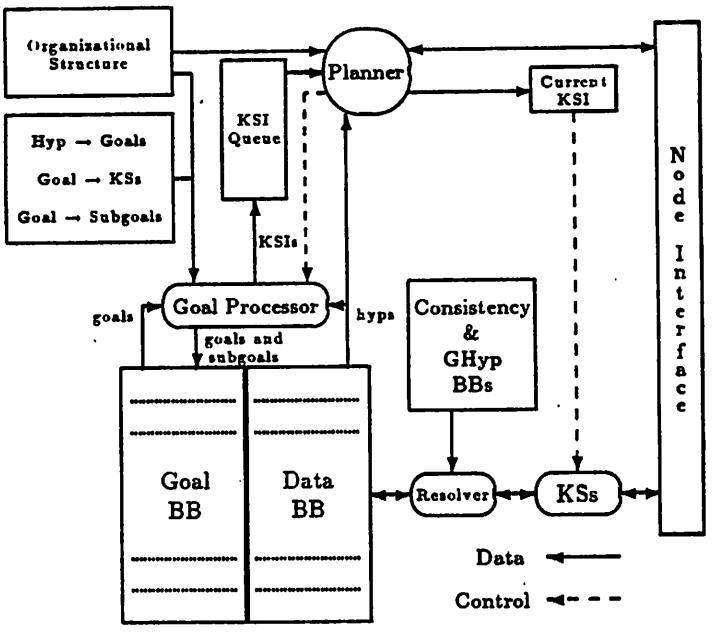


Figure 4: The DVMT Node Architecture.

is placed into service. This can again require changes to the blackboard representation to maintain high performance under operational conditions.

Before describing how the DVMT's blackboard implementation was tuned using GBB, we present a brief overview of the DVMT's problem-solving architecture, concentrating on its blackboard structure, blackboard objects, and blackboard retrieval characteristics. With this background in place, we describe our experiences tuning the DVMT operating on a two relatively small scenarios followed by the results of scaling these results to a larger scenario.

4.1 An Overview of the DVMT

The Distributed Vehicle Monitoring Testbed (DVMT) simulates a network of blackboard-based problem solving nodes working on the vehicle monitoring task. The objective of the network is to generate an answer map containing the identity and movement of vehicle patterns based on passively sensed acoustic data. Each network node is a complete Hearsay-II architecture [7] with KSs and blackboard levels appropriate for the task of vehicle monitoring. The basic control components of Hearsay-II have been augmented by goal-processing and planning capabilities (Figure 4). In this paper, we concentrate on the major blackboard components: the data, goal, consistency, and ghyp blackboards.

Hypothesized vehicle movements are represented by *hypotheses* placed on the *data blackboard* (D-BB). KSs perform the basic problem solving tasks of abstracting, extending, and refining these hypotheses. The D-BB is partitioned into four data abstraction levels: *signals* (containing minimally-processed sensory data), *groups* (representing harmonically-

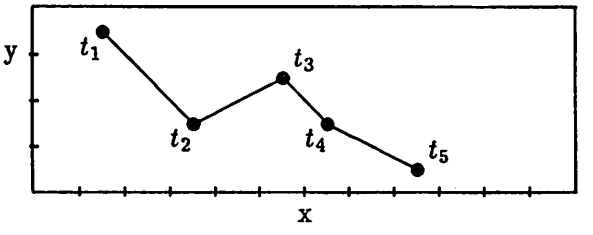
grouped signal hypotheses), *vehicles* (containing vehicle types hypothesized from related group hypotheses), and *patterns* (containing spatially-related vehicles, such as vehicles moving in formation). Each of these abstraction levels is split into a level for location hypotheses (which have one sensed position) and a level for track hypotheses (which have a compatible sequence of sensed positions over time) for a total of eight D-BB levels: SL, ST, GL, GT, VL, VT, PL, and PT.

KSs combine hypotheses to form more encompassing hypotheses on the same or higher levels. Decisions of which KSs to execute are made using a unified data- and goal-directed framework [12]. The control components of the DVMT (primarily the planner and goal processor) create *goals* on the *goal blackboard* (G-BB), which mirrors the 8-level organization of the D-BB. Each goal represents a request to create a one or more hypotheses on the D-BB within the (corresponding) area covered by the goal. KSs serve as the "actions" for achieving goals on the G-BB.

In addition to the D-BB and G-BB, the DVMT includes two "hidden" blackboards for instrumentation. The *consistency blackboard* (C-BB) contains hypotheses representing the correct solution hierarchy as precomputed from the input data. This oracle is invisible to the KSs and control components, but is used to evaluate the developing solution by simulation measurement tools. The *ghyp blackboard* (GH-BB) contains a complete centralized set of the sensory data. Again these hypotheses are only used by measurement tools.

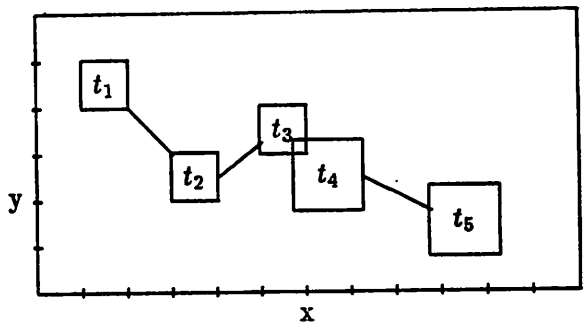
The details of hypotheses and goal objects are also important for tuning the application, and we briefly describe the structure of each.

A hypothesis on the D-BB, C-BB, or GH-BB has the following major attributes: one or more *time-locations* (the vehicle's sensed location at successive points in time), an *event-class* (the frequency classification or vehicle identity information), and a *belief* (the confidence in the accuracy of the hypothesis). The time-location structure of a hypothesis is represented in GBB as a *composite* unit[8] containing series of connected (x,y) points along the *time* dimension:



A goal has the following major attributes: one or more *time-regions* (areas of desired problem solving activity), a set of *event-classes*, and a *rating* (an estimate of the importance of achieving this goal). The time-region structure of a goal is represented as a composite unit containing series of connected (x,y) regions along the *time* dimension:

[8] GBB's blackboard objects.



All DVMT levels are implemented as GBB *spaces* with dimensions *time*, *x*, *y*, and *event-class*. (*Belief* and *rating* would also be useful dimensions but these attributes were not represented as dimensions in the current implementation of the DVMT.) *Space dimensionality* is central to GBB. It provides a *metric* for positioning units on the blackboard in terms that are natural to the application domain. Units are viewed as occupying some n-dimensional extent within the space's dimensionality. Application code can create and retrieve units according to the dimensions of spaces, without regard to the underlying implementation of the blackboard structure [13]. Dimensional references, however, contain enough information when combined with information about the structure of the blackboard to allow efficient retrieval code to be generated.

4.2 Experimental Background

Access to the DVMT provided the opportunity to empirically evaluate the performance of the DVMT simulator, given differing specifications for the blackboard database implementation. We selected a "typical" single problem-solving node scenario and created three configurations (each one increasingly complex) for experimentation. The first configuration (which will be labeled C1) had a reduced amount of sensory noise and a reduced grammar. The second configuration (C2) had a reduced amount of sensory noise and a full grammar. The third configuration (C3) had all the sensory noise and reduced grammar. The less complicated versions (C1 and C2) required significantly less processor time, and allowed us to run more tuning experiments.

The domain of these experiments was limited to the blackboard implementation strategies provided by GBB, and the performance comparisons are between GBB's various strategies. Considerable effort has been spent optimizing GBB's database machinery, and even GBB's default blackboard implementation strategy results in "reasonable" performance when fewer than 15–20 units reside on a blackboard level.[9]

We emphasize that identical processing occurs in all the experimental tests within an experiment suite. The input data is identical, KSs run in the same sequence, locate the same hypotheses, produce the same output, the control components make the same decisions, and so on. Furthermore, the abstract representation of the blackboard (its decomposition into spaces), the dimensionality of each space, and the unit retrieval pattern specifications remained constant. The *only* variable is the blackboard database machinery used by GBB to store and retrieve blackboard units.

[9]Due to its size, it is impractical to recode the DVMT to obtain performance measurements for a non-GBB-based implementation.

Our experiments concentrated on how hypotheses are stored on the D-BB, C-BB, and GH-BB and how goals are stored on the G-BB. Intuition led us to expect that when a small number of units were to be created on a blackboard level, a simple "push them on a list" implementation would be best due to its low overhead. When a large number of units were created on a space and numerous retrievals were performed on them, a more complex "dimensional-metric-based" implementation was appropriate. Finally, we expected that hypotheses and goals would have different balances in their storage strategies because hypotheses are composites of points while goals tended to be overlapping composites of (x,y) regions. (This expectation proved false.)

We began by analyzing each configuration's blackboard interaction statistics. GBB can provide the number and types of units created on each space, the number of insertion and retrieval operations performed on each space, and the time spent on these operations. The numbers of hypotheses and goals created on each blackboard space are as follows:

C1: Number of Blackboard Objects

Level	C-BB	D-BB	G-BB	GH-BB
SL	64	192	0	192
ST	4	0	0	
GL	32	264	96	
GT	2	0	0	
VL	16	44	44	
VT	1	164	362	Total
PL	16	0	0	2132
PT	1	188	450	

C2: Number of Blackboard Objects

Level	C-BB	D-BB	G-BB	GH-BB
SL	64	192	0	192
ST	4	0	0	
GL	32	264	96	
GT	2	0	0	
VL	16	44	44	
VT	1	164	362	Total
PL	16	0	0	2738
PT	1	352	892	

C3: Number of Blackboard Objects

Level	C-BB	D-BB	G-BB	GH-BB
SL	64	2176	0	2176
ST	4	0	0	
GL	32	342	1088	
GT	2	0	0	
VL	16	57	57	
VT	1	234	455	Total
PL	16	0	0	7628
PT	1	297	610	

The number of KS executions required to find the solution in each configuration is:

Configuration	KSIs
C1	743
C2	906
C3	1672

In all configurations, few units are created on the ST, GT, and PL levels. This is because the control components were instructed to restrict the synthesis path to the SL, GL, VL, VT, PT levels.

The number of blackboard unit retrieval operations is also important for tuning. For the each configuration, GBB reports the following operation counts. The tables show the number of retrieval operations for each space followed by the percentage of the total number of retrievals, enclosed in parentheses.

C1: Number of Blackboard Retrievals

Level	C-BB	D-BB	G-BB	GH-BB
SL	680 (3)	1344 (6)	192 (1)	556 (3)
ST	4 (0)	0 (0)	368 (2)	
GL	1184 (6)	800 (4)	456 (2)	
GT	2 (0)	0 (0)	504 (2)	
VL	1204 (6)	338 (2)	348 (2)	
VT	872 (4)	1439 (7)	712 (3)	Total
PL	16 (0)	0 (0)	44 (0)	21,228
PT	5343 (25)	2620 (12)	2202 (10)	

C2: Number of Blackboard Retrievals

Level	C-BB	D-BB	G-BB	GH-BB
SL	680 (2)	1344 (3)	192 (0)	556 (1)
ST	4 (0)	0 (0)	368 (1)	
GL	1184 (3)	800 (2)	456 (1)	
GT	2 (0)	0 (0)	504 (1)	
VL	1204 (3)	338 (1)	348 (1)	
VT	872 (2)	2604 (7)	712 (2)	Total
PL	16 (0)	0 (0)	88 (0)	38,551
PT	16839 (44)	5134 (13)	4306 (11)	

C3: Number of Blackboard Retrievals

Level	C-BB	D-BB	G-BB	GH-BB
SL	6624 (5)	16552 (13)	2176 (2)	5512 (4)
ST	4 (0)	0 (0)	4132 (3)	
GL	18536 (15)	2162 (2)	2530 (2)	
GT	2 (0)	0 (0)	672 (1)	
VL	3310 (3)	484 (0)	430 (0)	
VT	2912 (2)	2392 (2)	994 (1)	Total
PL	16 (0)	0 (0)	57 (0)	126,873
PT	45983 (36)	6058 (5)	5335 (4)	

Each retrieval operation involves a composite four-dimensional pattern in *time*, x, y, and *event-class*. In addition, the DVMT provides additional procedural filtering code to GBB's retrieval process. In our experiments, the time required by these filters is considered part of the retrieval time.

There are two things to note about these numbers. First, almost half the retrieval operations are from the C-BB (used in performance monitoring) but there are relatively few units stored on the C-BB. Therefore, a simple, "low-overhead" strategy is appropriate for representing the C-BB. Second, the distribution of retrieval operations on the D-BB and G-BB shifts dramatically from the filtered case to the complete case. (Surprisingly, we found the retrieval characteristics of hypothesis and goals to be very similar, and the representation strategy that worked well for one also worked well for the other.)

4.3 The Experiments

We began our tuning experiments by running the DVMT on configuration C1 (the simplest scenario) using its "designed" blackboard database implementation: a single vector for the *time* dimension. This storage strategy had been intuitively selected (by the second author, based on a pre-GBB implementation of the DVMT) as providing a reasonable balance between retrieval time versus insertion time and storage space. As the experiments demonstrated, this intuition resulted in only mediocre performance—an indication of the importance of database flexibility and performance monitoring tools!

The second experiment ran C1 with the simplest storage strategy, storing all units on a space in a list. As expected, this resulted in even poorer performance. We then tried two vectors, x and y. This gave a dramatic performance improvement, reducing the total execution time by more almost half compared to the baseline "list" strategy. Using three vectors *time*, x, and y resulted in a further 5% reduction in execution time.

We ran many additional experiments (approximately 90) using different strategies for each space and each type of unit. The best strategy was *time*, x, and y as a three dimensional array. The total execution time in the best case was one half that of the worst case (the simple "list" strategy). Even more dramatic is the decrease in the execution time due to blackboard operations. In the best case blackboard operations took only 10:23, whereas in the worst case blackboard operations took 30:01.

Table 1 summarizes the most interesting C1 experiments. Each experiment is identified by the storage strategy used. A list of letters indicates that each dimension is stored in a vector of buckets. An additional level of parentheses indicates that those dimensions are grouped together into a multi-dimensional array of buckets. For example, `(t x y c)` indicates four vectors (one each for *time*, x, y, and *event-class*) while `(t (x y))` indicates one vector for *time*, and one 2-dimensional array for x, and y. In the table, all buckets for the *time* and *event-class* dimensions were unit width; buckets for x and y were of width 5.[10] Furthermore, each space in the C-BB was represented as a simple list `()`, which was the most effective strategy given its limited number of units.

The processing times are in minutes and seconds from a single run on an 8 MByte Texas Instruments Explorer II. Differences of 10–20 seconds are insignificant due to timer resolution. The processing time for performing non-blackboard activities in each experiment was approximately 6:40 (ranging from 6:20–6:58). Mean paging time was 9 seconds (8–12 seconds). The last column (in parenthesis) gives the percentage of time spent doing blackboard operations.

[10] We experimented with varying bucket sizes, but in these scenarios "reasonable" changes in bucket width had little effect on performance.

Experiment	Total Time	BB Time
((t x y))	17:21	10:23 (60)
(t (x y))	17:44	10:48 (61)
(c (t x y))	17:45	10:59 (62)
(t c (x y))	18:00	11:20 (63)
((x y))	18:33	11:43 (63)
(c (x y))	18:48	12:03 (64)
(t x y)	18:56	12:14 (65)
(t x y c)	19:13	12:19 (64)
(x y)	19:52	13:03 (66)
(x y c)	20:06	13:19 (66)
(t)	20:47	14:26 (69)
(t c)	21:28	14:40 (68)
(y)	23:37	17:07 (72)
(x)	23:42	17:08 (72)
(c)	36:19	29:51 (82)
()	36:20	30:01 (83)

Table 1: C1 Configuration Experiments.

Table 2 contains the results with the C2 configuration. Again 10–20 second differences are insignificant. In this set, the processing time for performing non-blackboard activities in each experiment was approximately 18:00 (17:07–18:54). Mean paging time was 36 seconds (34–42 seconds).

The results from C3, the most complex configuration are in Table 3. In this set, the processing time for non-blackboard operations was approximately 135:00 (ranging from 130:33–137:45). Mean paging time was 5:30 (4:50-7:08).

As the three sets of results show, tuning the blackboard representation results in even more dramatic performance improvements as the complexity of the configuration is increased. In C1 and C3 there are only a small number of event classes. In C1 the single vector *event-class* unit mapping is no faster then the simple "list" unit mapping. But, in C3, because of the increased amount of sensory noise the `(event-class)` mapping is 40% faster than the `()` mapping.

In some cases the overhead of using an additional dimension in the unit mapping is not worth it. For example, in C1 and C3, using *event-class* doesn't improve performance at all. Except for the single vector `(event-class)` mapping, any mapping that uses *event-class* does worse than the same mapping without *event-class*. This is because, in C1 and C3, there are very few *event-classes* so it doesn't provide any discriminatory power. In C2 adding *event-class* does improve performance noticeably.

Regardless of the mapping used, the time required to insert units on the blackboard was less then 1% of the total runtime. (The range was $\approx$0.1% for C3 up to $\approx$0.9% for C1.) The relatively small insertion cost was surprising, even to the implementers of GBB. Virtually all the blackboard time was spent in retrieval. In fact, 80–90% of the retrieval time was spent in the pattern based filtering step.

The reduced execution time is the direct result of greatly reducing the number of units

Experiment	Total Time	BB Time
(c (t x y))	37:08	18:15 (49)
(t c (x y))	37:56	19:15 (51)
(c (x y))	38:51	20:26 (53)
((t x y))	39:22	20:43 (53)
(t (x y))	40:09	21:56 (55)
(t x y c)	40:11	21:40 (54)
(x y c)	41:14	22:58 (56)
((x y))	41:51	23:27 (56)
(t x y)	43:06	24:55 (58)
(t c)	43:51	25:29 (58)
(x y)	45:11	26:58 (60)
(t)	49:22	31:04 (63)
(y)	54:59	37:05 (67)
(x)	55:11	37:28 (68)
(c)	68:43	51:37 (75)
()	84:40	67:21 (80)

Table 2: C2 Configuration Experiments.

Experiment	Total Time	BB Time
((t x y))	203:18	65:34 (32)
(t (x y))	205:09	68:09 (33)
(c (t x y))	205:42	68:07 (33)
(t c (x y))	207:00	70:35 (34)
(t x y)	207:22	70:43 (34)
(t x y c)	208:52	72:15 (35)
(t)	210:06	89:57 (43)
((x y))	212:46	76:53 (36)
(x y c)	213:38	77:43 (36)
(c (x y))	214:25	78:46 (37)
(x y)	216:11	79:59 (37)
(t c)	218:15	82:37 (38)
(x)	246:54	111:04 (45)
(y)	248:19	112:06 (45)
(c)	353:15	222:41 (63)
()	594:55	493:12 (83)

Table 3: C3 Configuration Experiments.

that must be considered by the three filtering steps. The following table illustrates the effect of different mappings on the number of units retrieved by the primary retrieval step. The first column shows the average number of units returned by the primary retrieval for the entire run. The second column shows the total time spent in the pattern based filtering step. These numbers are for the PT level of the G-BB for configuration C1. At the end of the run there were 892 units on this space. On average 26.25 units passed the pattern based filtering step.

Experiment	Primary Retrieval Count	Pattern Filtering Time
(c (t x y))	43.81	7:16
(c (x y))	62.15	8:48
(t)	150.65	14:59
(x)	228.93	19:42
(c)	261.89	26:34

Using the best mapping, `(c (t x y))`, more than 50% of the units in the candidate set were in the final set of retrieved units. While the worst mapping, `(c)`, produced candidate sets in which only 10% of the units survived the filtering steps. Note that the time spent in the pattern based filtering step is proportional to the number of units in the candidate set. The best mapping spent 70% less time in the pattern based filtering step compared to the worst mapping.

5 Guidelines for Blackboard Representation

As our performance tuning experiments indicate, selecting an appropriate blackboard representation (the unit mapping) can have a significant effect on application performance. Here are some general guidlines or hints on acheiving the best performance in GBB by tuning the blackboard representation.[11] Many of them are simple common sense. As with most efficiency techniques, the law of diminshing returns applies. For example, changing from a simple list implementation to a single vector implementation will improve search speeds by 50% to 90%.

- When a small number of units will be created on a space the simple "list" implementation is best because of its low overhead. The threshold is somewhere between 5–8 units with complicated indexes (i.e., composite indexes with several elements) and 20–25 units with simple scalar indexes.
- When all units tend to span the entire range of a space dimension that dimension should be left out of the unit mapping.
- If one dimension evenly divides the units on a space it should be included in the unit mapping even if it only has a few distinct values.
- If the units have complicated indexes then the increased selectivity of the grid (n-dimensional array) is well worth the additional space required. Searches on a 3-dimensional grid have run up to 40% faster than equivalent searches on three vectors.

[11] These are based on experiments run on a Texas Instruments Explorer II workstation. Additional experiments are required to see if our findings extend to other lisp imlementations.

- Conversely, if the units have scalar indexes then the partition strategy (n vectors) tends to be almost as good as the grid strategy.
- If the units are distributed on a diagonal with respect to the space dimensions, consider using indexes that are a transformation (e.g., rotation) of the "actual" values. This can be done efficiently using the index structure mechanism in GBB. We are considering adding such transformations as a built-in feature of GBB.

6 Summary

Reducing the cost of blackboard retrieval can significantly increase the performance of blackboard-based applications. In the Generic Blackboard Development System (GBB) we have provided application developers the mechanisms for high-performance retrieval operations. The blackboard representation techniques used in GBB were selected to provide the best performance over a wide range of situations.

GBB also provides the tools to analyze the performance of blackboard operations in a particular application. We have demonstrated the effectiveness of adjusting the blackboard representation (the unit mappings) in improving blackboard performance. Tuning the DVMT by matching its blackboard database structure to its blackboard interaction characteristics resulted in significant performance improvements.

These results do not suggest that blackboard database optimization should replace the use of superior problem-solving knowledge or control capabilities as a means of enhancing performance. They do demonstrate, however, that improving blackboard interaction efficiency should not be neglected. The potential performance improvements due to the blackboard implementation are proportional to the ratio of blackboard interaction to KS (and control) processing.

Our experiences with the DVMT tuning process demonstrates the importance of obtaining detailed measurements of the insertion/retrieval characteristics of each space (and even within a space). These measurements can significantly augment "intuitive" decisions for blackboard implementation strategies and form an important component of a blackboard development environment.

References

[1] Jon L. Bentley and Jerome H. Friedman. Data structures for range searching. *Computing Surveys*, 11(4):397–413, December 1979.

[2] Richard D. Fennell and Victor R. Lesser. Parallelism in Artificial Intelligence problem solving: A case study of Hearsay II. *IEEE Transactions on Computers*, C-26(2):98–111, February 1977. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 106-119, Morgan Kaufmann, 1988.).

[3] Kevin Q. Gallagher, Daniel D. Corkill, and Philip M. Johnson. *GBB Reference Manual.* Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, GBB Version 1.2 edition, September 1988. (Published as Technical Report 88-66, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, September 1988.).

[4] Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray. GBB: A generic blackboard development system. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1008–1014, Philadelphia, Pennsylvania, August 1986. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 503–518, Addison-Wesley, 1988.).

[5] Victor R. Lesser and Daniel D. Corkill. The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, Fall 1983. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 353–386, Addison-Wesley, 1988 and in *Readings from AI Magazine: Volumes 1-5*, Robert Engelmore, editor, pages 69–85, AAAI, Menlo Park, California, 1988).

[6] Victor R. Lesser, Daniel D. Corkill, and Edmund H. Durfee. An update on the Distributed Vehicle Monitoring Testbed. Technical Report 87-111, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, December 1987.

[7] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.

[8] Hanan Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, June 1984.

[9] Steve Rowley, Howard Shrobe, and Robert Cassels. Joshua: Uniform access to heterogeneous knowledge structures or why joshing is better than conniving or planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 48–52, Seattle, Washington, July 1987.

[10] Stuart Russell. The compleat guide to MRS. Technical Report KSL No. 85-12, Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, California 94305, June 1985.

[11] Intellicorp. *KEE V3.1 Reference Manual*, 1987.

[12] Daniel D. Corkill, Victor R. Lesser, and Eva Hudlická. Unifying data-directed and goal-directed control: An example and experiments. In *Proceedings of the National Conference on Artificial Intelligence*, pages 143–147, Pittsburgh, Pennsylvania, August 1982.

[13] Daniel D. Corkill, Kevin Q. Gallagher, and Philip M. Johnson. Achieving flexibility, efficiency, and generality in blackboard architectures. In *Proceedings of the National Conference on Artificial Intelligence*, pages 18–23, Seattle, Washington, July 1987. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 451–456, Morgan Kaufmann, 1988.).

[14] Daniel D. Corkill and Kevin Q. Gallagher. Tuning a blackboard-based application: A case study using GBB. In *Proceedings of the National Conference on Artificial Intelligence*, pages 671–676, St. Paul, Minnesota, August 1988.

[15] Kevin Q. Gallagher and Daniel D. Corkill. Blackboard retrieval strategies in GBB. Technical Report 88-39, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, December 1988.