Prototyping Bubba, A Highly Parallel Database System

HARAN BORAL, WILLIAM ALEXANDER, LARRY CLAY, MEMBER, IEEE, GEORGE COPELAND, MEMBER, IEEE, SCOTT DANFORTH, MICHAEL FRANKLIN, BRIAN HART, MARC SMITH, AND PATRICK VALDURIEZ

Abstract—**Since 1984, the goal of the Bubba project at MCC has been to design a scalable, high-performance and highly available database system that will provide significant cost/performance advantages over conventional mainframes in the 1990's. The design process has been an iterative one, cycling through design, modeling, and prototyping in progressive detail. The current Bubba prototype runs on a commercial 40-node multicomputer and includes a parallelizing compiler, distributed transaction management, object management, and a customized version of UNIX. This paper describes the current prototype and discusses of the major design decisions that went into its construction. The lessons learned from this prototype and its predecessors are presented.**

Index Terms—**Complex object management, database operating system, database programming language, database system performance, database system prototype, dataflow execution, parallel database system.**

I. Introduction

Bubba is a highly parallel computer system for data-intensive applications, which has been designed and prototyped at MCC. The basis of the Bubba design is a scalable *shared-nothing* architecture which can scale up to thousands of nodes. Data are *declustered* across the nodes (i.e., horizontally partitioned [33], [37] via hashing or range partitioning) and operations are executed at those nodes containing relevant data. In this way, parallelism can be exploited within individual transactions as well as among multiple concurrent transactions to improve throughput and response times for data-intensive applications.

Much of the Bubba design and implementation effort has gone into developing the technology necessary to efficiently manage and exploit parallelism. This effort can be divided into four separate areas:

Data Placement—Bubba was designed for "data-intensive" applications, where the data are too large and accessed too frequently to be shipped between nodes for processing. Instead, operations are executed at the nodes which contain the data. As a consequence, the placement and declustering of data across the nodes of the system directly determines the load across the system. Proper data placement is crucial to Bubba's performance, and must be periodically adapted to changes in workload access patterns.

Automatic Parallelization—An important requirement for Bubba was to allow transaction programs to be written using a centralized execution model (i.e., as if all of the data were stored on a single node). The Bubba compiler automatically decomposes monolithic transaction programs into multithreaded parallel programs.

Dataflow Control—In most dataflow machines, each dataflow operation executes on a single hardware unit. In Bubba, each dataflow operation may execute in parallel on possibly many nodes. The nodes that participate are determined by the data required to perform the operation. When data are sent collectively from one operation to another, *dataflow control* is needed to tell each receiving node the identities of the sending nodes and the number of messages to expect. The challenge in efficient dataflow control is to identify the sending nodes and inform the receiving nodes while keeping overhead to a minimum.

Data Recovery Techniques—Bubba supports applications that require high availability. However, as the number of nodes is increased, node failures will be more frequent. We have developed a number of techniques to allow the system to continue processing in the presence of node failures and to quickly bring substitute nodes back on-line after a failure.

While the main thrust of the project has been parallelism, the Bubba design includes novel approaches to the following important areas of database systems:

Database Programming Languages—In the early stages, Bubba was part of a larger project called ADBS (Advanced Database System) whose intent was to marry a logic-programming language called LDL [36], [46] with a high-performance parallel implementation. An intermediate language called FAD [9], [20] was designed. Later, the larger project divided into two separate projects: an LDL project with emphasis on compiling logic programs in LDL, and the Bubba project with emphasis

Manuscript received October 15, 1989; revised December 10, 1989.
H. Boral was with the Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759.
W. Alexander and S. Danforth are with the Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759.
L. Clay is with Tandem Computers, Austin, TX 78758.
G. Copeland and M. Smith are with IBM, Austin, TX 78758.
M. Franklin is with the Department of Computer Sciences, University of Wisconsin, Madison, WI 53706.
B. Hart is with BULL, France.
P. Valduriez is with INRIA, Rocquencourt, France.
IEEE Log Number 8933793.

on parallelizing FAD. FAD includes a significant extension of relational functionality in both its data modeling and general-purpose programming capability. The main intent of improved data modeling was to allow expensive joins to be avoided using arbitrarily nested structures. The main intent of improved general-purpose programming was to allow more of an application to be compiled and executed directly in Bubba than is currently possible in conventional (e.g., embedded SQL) systems, thereby avoiding excessive data movement between the programming-language and database systems.

Object Management—The concept of a *single-level store* was fully exploited. A single-level store allows all data to be uniformly represented in a large virtual address space, regardless of whether it is transient versus persistent or whether it lives in memory versus disk. The intent was to improve performance and simplicity by avoiding data translation between different representations, to increase the amount of compile-time versus run-time support, and to have a single kind of buffer manager. Storage management techniques were developed to allow objects to be arbitrarily structured or sized.

Operating Systems Support—The intent of BOS (Bubba Operating System) was to have better operating system support for several of the object management functions, including single-level store, and MMU-assisted locking and workspace management. In addition, hooks are provided that allow the database system code to control scheduling, paging, and locking policies.

Many aspects of Bubba have been described elsewhere [3]–[5], [9]–[11], [13]–[16], [18]–[20], [27]–[33], [40], [47]–[51]. Because of the aggressive nature of the project, we decided at the beginning that we would need quantified performance goals for a specific workload and a design process driven by those goals. In this paper, we concentrate on the process of design, modeling, and prototyping the Bubba system.

Section II presents a brief overview of the Bubba system goals and design. Section III describes the early phases of the project including modeling and an initial prototype. Section IV describes the current 40-node prototype. Section V presents some initial results of speedup and scale-up experiments performed on the prototype. Section VI summarizes the more important lessons we learned.

II. Bubba Design Overview

A. Design Goals

The overall goal of the Bubba project has been to design a system for current and future data-intensive applications that has a significant cost–performance improvement over conventional mainframe-based database systems in the 1990's timeframe. While significant improvement for conventional transaction processing workloads (e.g., Debit–Credit [7]) is desired, the most dramatic improvements are to be in the support of "knowledge-based" transactions, which access and analyze large amounts of data.

In addition to the goal of good cost–performance, we had these goals.

- The system must have modularly-scalable performance. That is, it can be continuously expanded in throughput (as well as storage capacity) up to the "high-end" by adding more hardware modules.
- The system must have more functionality than relational systems. This functionality includes an improved data modeling capability to handle complex objects in an object-oriented style, and a general-purpose programming capability to allow more flexibility in deciding whether to implement a function in Bubba versus a host system.
- The system must have high reliability and availability. Reliability means that the system does not make unrecoverable mistakes in spite of component failures. Availability means that the system continues to work with adequate performance in spite of component failures. Our goal is to be at least as good as conventional systems using both mirroring and checkpoint-and-log recovery techniques.

B. Bubba Hardware Architecture

Bubba is a shared-nothing multiprocessor, i.e., the processors do not share memory or disks [42]. We chose this architecture primarily because it is the only architecture that can scale up to the performance levels dictated by our goals, and secondarily because it provides better reliability and availability by isolating faults.

This hardware organization is illustrated in Fig. 1. Bubba contains three types of nodes: *interface processors* (IP's), *intelligent repositories* (IR's), and *checkpoint-and-log IR's* (CIR's). These nodes are connected via a scalable message-passing interconnect, such as a hypercube. The IP's provide communication with external host machines and coordinate execution of user requests. The IR's collectively store the database and perform most of the work in executing transaction programs. The CIR's maintain database checkpoints and update logs for data recovery from IR failure. The majority of the nodes in a Bubba system are IR's. An IR minimally consists of a processor (p), a large amount of main memory (m), and a disk (d). The large memory improves performance by allowing heavily accessed persistent data (including cluster indexes), transient data, system tables, and programs to be cached in memory. Although IR's are shown to consist of a single processor and a single disk, they could in fact have a number of these.

The shared-nothing architecture of Bubba allows each IR to function in many ways as an independent centralized database system. (This feature was exploited when building the prototype systems.) Each IR contains fragments of database relations, determined by hash or range partitioning. Each IR applies program operations to its database fragments.

C. FAD Language

The current Bubba interface is the FAD language [20]. FAD significantly extends relational database functionality by providing

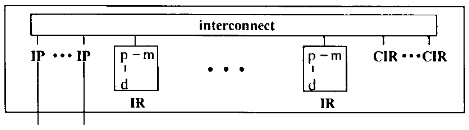


Fig. 1. Bubba hardware organization.

• complex objects, consisting of sets, tuples, and atoms, which can be nested to an arbitrary level

• the notion of object identity which allows objects to be referentially shared (i.e., objects can be graph-structured)

• data manipulation functions that are oriented to accessing nested sets and tuples

• control primitives, such as while-do and if-then-else, to support general purpose programming.

FAD treats transient and persistent data uniformly. *Transient data* are visible only to the transaction that creates it and live only for the lifetime of that transaction. *Persistent data* are visible to multiple transactions and exist beyond the life of any single transaction. A FAD object becomes persistent if it is reachable from (i.e., nested within) a special persistent root tuple called *db*. The root *db tuple* is a FAD tuple whose attributes are the *basesets* (also referred to as *base relations*) of the database.

By uniformity in FAD, we mean that persistence is orthogonal to type (i.e., an object of any type may become persistent) and that FAD operations can be applied to objects regardless of whether they are persistent or not [8]. Atomicity, concurrency control, and recovery issues are, for the most part, hidden from the FAD programmer.

Although data are declustered across some or all of the IR's in the system, FAD presents single-site semantics to the FAD programmer (i.e., the database appears to be centralized). The FAD compiler performs the mapping between the single-site semantics of FAD and the multinode, shared-nothing model supported by Bubba.

D. Distributed Execution Model

The distributed execution model of Bubba is based on dataflow concepts instead of remote-procedure calls, because dataflow allows a much higher degree of parallelism.

The FAD compiler translates a FAD program to PFAD (Parallel FAD), a language which is an extension of FAD and in which decisions concerning distributed execution on the declustered shared-nothing architecture of Bubba are explicitly expressed. PFAD uses the notion of program *components* which communicate via messages on *dataflow arcs* [27], [31], [32]. The purpose of a component in PFAD is to group PFAD actions that require access to the same data. The PFAD program indicates *logically* where each component will execute on Bubba, in terms of the data it will use (e.g., "execute component 1 on baseset db.S"). The binding to physical IR's is performed at run-time, determined by the current declustering and the data objects that are selected.

This parallel execution model allows three types of parallelism to be exploited:

• *intertransaction parallelism*—multiple transaction programs can execute concurrently.

• *intratransaction parallelism*—PFAD components of a single transaction may execute concurrently (restricted only by dataflow dependencies).

• *intracomponent parallelism*—a single component may execute concurrently on multiple IR's.

Fig. 2 illustrates the distributed execution of an example query transaction. The physical schema consists of three basesets and is similar to the familiar supplier-parts schema, except that it exploits the nested capabilities of FAD and includes inverted files. The query, called "CityParts," finds the description of parts supplied by any supplier in a specific city (in this case "Austin"). The PFAD query contains

1) a select on the inverted-file baseset db.Suppliers_Scity to get the S#'s for "Austin"

2) a join of this result with baseset db.Suppliers to get the Item#'s

3) a join of this result with baseset db.Items to get the tuples that form the final result.

Each of these three operations corresponds to a component whose *home* is the set of IR's containing one of the component's operand basesets. If there is more than one operand baseset, as in the join components, the FAD optimizer chooses the home such that the minimum amount of data are shipped between IR's.

The data placement of the three relations (as shown in the figure) is known in the *global directory*, which is replicated in all IR's. The global directory also knows how the tuples of each relation are declustered across each home (by hash or range partitioning on key values). When given a key value of a tuple, the global directory returns the identity of the IR which stores the tuple.

The transaction execution steps are as follows.

1) The execution begins by creating a *transaction coordinator* (TC) to coordinate the transaction, which resides in the IP that received the execution request from the user. The TC determines from the global directory that IR0 contains the tuple of db.Suppliers_Scity that has "Austin" as a key, and sends a message to IR0 to begin the dataflow execution.

2) Component 1 (the select) has only one *thread* (a lightweight process in Bubba) because it executes in only one IR. Component 1 executes the select and saves the S#'s. It then determines from the global directory that IR2 and IR4 contain the tuples of db.Suppliers which have the join values as keys, and sends a message to each of these two IR's. Each message to an IR contains only the Component 1 S#'s that can join with the corresponding tuples of db.Suppliers in that IR.

3) Component 2 (the first join) has two threads because it executes in two IR's. Each thread of Component 2 executes the partial join and saves the Item#'s. It then determines from the global directory that IR0, IR3, and IR4 contain the tuples of Items which have the join values as

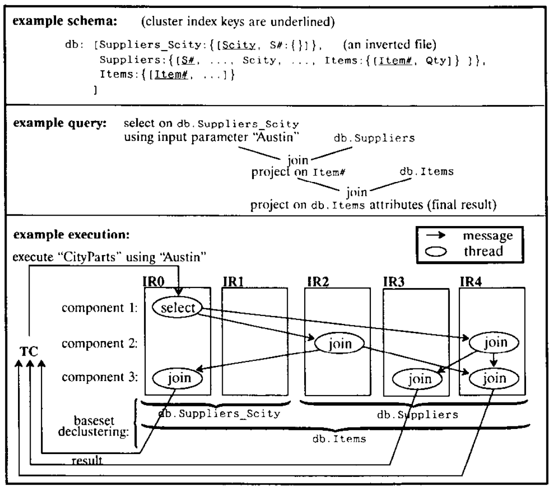

example schema: (cluster index keys are underlined)
db: [Suppliers_Scity:{[Scity, S#:{}]}, (an inverted file)
Suppliers:{[S#, ..., Scity, ..., Items:{[Item#, Qty]}]},
Items:{[Item#, ...]}
]
example query: select on db.Suppliers_Scity using input parameter "Austin"
db.Suppliers
join
project on Item#
db.Items
join
project on db.Items attributes (final result)
example execution:
execute "CityParts" using "Austin"
message
thread
IR0
IR1
IR2
IR3
IR4
component 1:
component 2:
component 3:
select
join
join
join
join
join
TC
baseset declustering:
db.Suppliers_Scity
db.Suppliers
db.Items
result

Fig. 2. Execution of an example query transaction.

keys, and sends a message to each of these three IR's. Each message to an IR contains only the Component 2 Item#'s that can join with the corresponding tuples of db.Items in that IR.

4) Component 3 (the second join) has three threads because it executes in three IR's. Each thread of Component 3 executes the partial join, saves the tuples of db.Items, and sends that as final results to the TC.

5) The TC sends a commit message to all involved nodes (telling the nodes to free up clan resources and locks) and then relays the final result to the user.

Note that the example contains only foreign-key joins, which are most efficiently performed in the baseset homes shown in the figure. Joins over nonkey attributes can be performed efficiently in parallel by first redistributing the two operand basesets across a *symbolic home* of arbitrary IR's using the join-attribute values for partitioning, and then performing a local join at each IR (similar to techniques described in [21], [23], [24]). The number of IR's in a symbolic home is chosen by the FAD optimizer at compile-time; the actual IR's are chosen (arbitrarily) at run-time and represented in a transaction-specific *symbolic home directory.* The symbolic home directory maps tuples onto a symbolic home's IR's via hashing.

E. Run-Time Support

The Bubba run-time support for execution of transactions can be divided into three main subsystems:

Distributed Execution—manages the execution of transactions across multiple nodes.

Object Management—implements the semantics and storage management for FAD complex objects.

Bubba Operating System (BOS)—A customized operating system that provides specialized database functions.

1) Distributed Execution: The distributed execution software manages the loading, activation, execution, and termination of transaction programs in Bubba's distributed shared-nothing environment. Transaction programs can be dynamically loaded and activated at run-time, or preloaded and preactivated in anticipation of execution. Each has its advantages under different conditions [3], [4]. The goal of dynamic loading and activation is to avoid the unnecessary overhead of messages, startup, and termination of components on IR's which do not contain data relevant to the operation, and therefore do no useful work. A component is dynamically loaded or activated on an IR at the time that the input data for the component arrive at the IR. The goal of preloading and preactivation is to avoid run-time delays. Combinations of the dynamic or anticipatory techniques are chosen to optimize the execution of a program.

Threads of components communicate using send and receive operations. A receive operation blocks until messages have been received from all of the IR's executing components that send data messages to the receiving component. Determining when all messages have been received is complicated by the fact that the identities of the sending IR's are often determined at run-time by *associative routing*, i.e., using the value of a data object to determine its appropriate destination by consulting the global directory or symbolic home directory. As mentioned in Section I, the process of coordinating sends and receives is referred to as *dataflow control.* Bubba uses three dataflow control methods, each of which performs best under different circumstances. These dataflow control methods are described in [3] and [4], and are illustrated in Section IV-D.

2) Object Management: The object management software provides efficient allocation, access, modification, and garbage collection of both persistent and transient FAD objects. The goals of the object management software are to support the object model of FAD, and to minimize disk I/O and data conversions by exploiting the novel features of BOS. For example, the object management software implements FAD's uniform treatment of persistent and transient data by using identical data structures within persistent and transient virtual address spaces provided by BOS.

The object management software supports cluster indexes for associative access on sets, tuples, and large atoms. Concurrency control for indexes is implemented using nontwo-phase locking techniques (i.e., semaphores); for data, it is implemented using implicit locking provided by BOS. *Boxing* (garbage collection and clustering) of persistent objects is performed to achieve high access locality. In Bubba, simple boxing is performed at the end of every transaction that updates the persistent space. Additional *background boxing* (for optimized clustering) can be performed on objects whenever an IR would otherwise be idle.

3) Bubba Operating System: BOS is a tailored operating system, which provides specialized features for supporting distributed execution and object management. Some of these features are the following.

- *Single-Level Store*—BOS provides the single-level storage abstraction in which the entire persistent space of an IR is mapped into the virtual address space of each process executing a transaction in the IR.

• *Locking and Workspace Management*—BOS uses conventional virtual memory management hardware to provide implicit page locking and transaction workspaces.

• *Two Page Sizes*—BOS provides support for small memory pages and large disk blocks. A small memory page size (e.g., 512 bytes) avoids poor memory utilization, complex and time-consuming object locking, and time-consuming page copying. A large disk block size (e.g., a disk track) avoids poor disk arm utilization, large system tables, and large cluster indexes. Most conventional systems try to compromise by choosing a single medium-sized page (e.g., 4K bytes), but instead suffer all of the above problems to varying degrees.

• *Lightweight Process Threads*—BOS provides multiple lightweight threads of execution within a process, which allows PFAD components to be dynamically activated and executed without excessive overhead.

• *Communication Primitives*—BOS provides send and receive primitives that are used to implement the dataflow semantics of PFAD programs. All messages are implemented by unacknowledged datagrams, for efficiency; acknowledgments can be programmed explicitly when necessary.

While BOS is not a distributed operating system (i.e., it does not provide distribution transparency), it provides the support required to implement distributed processes in Bubba.

III. Early Models and Prototyping

A. Iterative Performance-Driven Design Process

Given our ambitious performance goals, we adopted an iterative design process incorporating both modeling and prototyping feedback, in which the design of Bubba evolved over several years. Performance predictions from models as well as measurements from prototypes helped us choose between design alternatives, or at least understand the tradeoffs among the alternatives if the choice is workload-dependent. In a few circumstances, the analyses caused us to revise previous design decisions.

It should be emphasized that we did not rely on either modeling or prototyping exclusively. The prototypes provide "real" measurements, but only for small system configurations. We recognized the need to develop models, parameterized using prototype measurements, to demonstrate system performance in very large configurations.

The earliest simplest models were developed as a quick means of "disaster avoidance." The system design was characterized at a high level in terms of basic resources such as CPU's, disks, memory, and interconnect. Performance predictions were made using ballpark estimates for resource rates of service and capacities, in the style of [39]. The goal was to expose critical performance problems with the overall architecture, even with best-case assumptions about the system resources and workload. In one case, a simple model dramatically showed the importance of data placement to good large-scale performance.

As the design progressed, more detailed models were developed to examine specific parts of the system design, such as the interconnect, dataflow control protocols, concurrency control, and recovery. These models were sometimes tailored to the specific experiment, using analytic modeling, simulation, or a combination of both.

A "full" simulation of Bubba which models the final system design was constructed in order to more accurately predict the scalability of Bubba's performance over the entire range of configuration sizes (up to 1024 nodes). However, for various nontechnical reasons, the model was put "on hold" for some time, and as of this writing is still being tested.

The prototyping effort was also iterative. A first prototype was intended to help us flesh out an appropriate system software design and identify important performance issues, with subsequent prototypes to provide useful measurements. The first prototype and the lessons we learned from it are discussed later in this section.

B. The Order-Entry Workload

To drive our performance analyses, we developed a gedanken, yet realistic, characterization of an order-entry system application, enhanced with decision-support (or "knowledge-based") programs. This workload was considered representative of those that might run on Bubba. Different experiment workloads are composed by specifying a desired mix of five order-entry transactions: *New-Order*, *Order-Shipped*, *Payment*, *Suggested-Order*, and *Store-Layout*. The order-entry transactions and their workload characterizations are further described in [5]. The current order-entry physical database consists of eight basesets containing information on items, customers, orders, etc. The five transactions are summarized as follows.

• *New-Order* records a customer's order for an average of ten different items after the customer's new outstanding balance is checked against the customer's credit limit.

• *Order-Shipped* records the shipping date for an order and generates an invoice for the customer.

• *Payment* records the payment date for an order and adjusts associated customer and salesperson sales totals. Payment is most similar to the Debit–Credit transaction [7], although Payment requires somewhat more work and has less potential parallelism due to dataflow dependencies.

• *Suggested-Order* infers the number of items to order from suppliers, to keep a warehouse sufficiently stocked.

• *Store-Layout* assists a customer (e.g., a store) in configuring the layout of items on the shelves in the store in an attempt to maximize customer profit.

For each transaction type in the current order-entry workload, Fig. 3 illustrates its relative frequency in our most often used mix, and its performance-dominant operation. The conventional transactions contain update op-

Transaction	Fraction In Mix	Dominant Operation	Start+Termination Overhead	Message Overhead
New-Order	0.332	simple update	O(1)	O(1)
Order-Shipped	0.332	simple update	O(1)	O(1)
Payment	0.332	simple update	O(1)	O(1)
Suggested-Order	0.001	large scan	O(N)	O(N)
Store-Layout	0.003	large N-M join	O(N)	O(N*M)

Fig. 3. Order-entry workload characterization.

erations involving a small number of tuples. The two decision-support programs, Suggested-Order and Store-Layout, are read-only queries, and contain a large scan and a large *N-M* join, where *N* and *M* refer to the number of nodes containing the two joined relations.

As part of the workload characterization, each of the order-entry transactions were coded in FAD and parallelized with each transaction represented by a dataflow graph. Each node in a dataflow graph represents a FAD operation (e.g., selection, join, insert, etc). Arcs between nodes represent transmission of intermediate results between operations and, therefore, may constrain the start time of an operation.

The five transactions in the order-entry suite are reasonably diverse, using different FAD operations and exhibiting different degrees of intratransaction and intracomponent parallelism. This diversity provided us with a wide range of test cases for experiments.

C. Analyses of Design Alternatives

Here, we summarize the results of several model-based analyses that influenced key parts of the Bubba design. Many of the simulations were done using a process-oriented simulation package, CSIM [38].

Data Placement—In a series of data placement experiments [15], an analytic model called FIRM [11] was used to predict the throughput performance of the order-entry workload running on a 1024-IR Bubba configuration. Using this model, we developed the data placement algorithm that assigns each relation to an appropriate number of IR's (i.e., the relation's home), in a way that balances the load for all the relations as evenly as possible across all the IR's. The model demonstrated the effect a particular data placement algorithm had on performance. One of our very first results showed how poor data placement can drastically affect performance due to poor load balancing. Using feedback from the model, the data placement algorithm was repeatedly refined to better use information about the "heat" (i.e., frequency of access) and size of each relation. The throughput performance achieved using the final algorithm described in [15] was more than two orders of magnitude better than earlier algorithms.

Process and Dataflow Control—In [3] and [4], program loading, activation and dataflow execution strategies were proposed for Bubba and analyzed. Cost formulas were developed to quantify the tradeoffs of preloading and dynamically loading program code, and show when it is more cost effective to cache a program in memory or store it on disk for reuse versus reloading the program over the interconnect. Finally, two approximate analytic models and a simulation model were used to evaluate the performance of three different dataflow control protocols in terms of throughput, response time, and number of packets sent. Each of the dataflow protocols differs in the number of messages sent and the number of times data must be copied before it reaches its destination. Using the models, we developed an algorithm that is used by the FAD compiler's optimizer to choose the most efficient protocol for each dataflow arc using selectivity and size information.

Interconnect—Early in the design of Bubba, we had considered the use of an "intelligent switch," which would provide hardware assistance for data and program routing, program control (e.g., if-then-else, while-do), concurrency control and data merging operations. After much analysis, it was evident that making the switch intelligent in these ways would not be cost-effective. Therefore, Bubba now only relies on a more conventional message-passing interconnect. While the Bubba design is not tied to any particular topology, our analyses show that a hypercube would provide ample performance which scales nicely as the system size grows, would provide fault tolerance, and can be easily packaged even for large configurations, all using hardware technology that is easily available today. We found that the same data declustering mechanism used to load balance the IR's also did an adequate job of load balancing the interconnect, so that hot spot problems often encountered in multistage interconnects were avoided. In another study [6], several hypercube routing algorithms were simulated, and their performance was compared in the presence of hot spots and faults.

Schema Design—Several order-entry schemas were designed. We found that performance could be significantly improved by nesting objects to avoid joins. One significant example of this was the Suggested-Order transaction, which could be transformed from a time-consuming large join into a much faster large scan. Several other examples included eliminating smaller joins. Interestingly, there was little performance advantage in the order-entry workload for the object sharing capabilities of FAD among persistent objects. However, we note that when creating a local transient object from a persistent one, direct object references can often be used to avoid copying the (perhaps large) persistent object.

Distributed Two-Phase Locking—A series of simulation experiments were performed to study two-phase locking performance in shared-nothing parallel machines like Bubba, ranging in size from 4 to 256 nodes [28]. An order-entry workload was used. The results showed that system throughput as limited by lock conflicts approaches the asymptotic conflict-free system throughput as the system size increases, boding well for larger system configurations. The results also demonstrated how performance of timeout-based deadlock resolution schemes can sometimes be highly sensitive to the choice of timeout values. The hottest relations rates in this study happened to be small, yielding higher conflict rates and making the timeout period more critical. In turn, we have considered the use of a separate global deadlock detector.

Safe RAM—In [18], we examined how *safe RAM* (i.e., nonvolatile and protected) can be implemented with conventional hardware technology and used effectively in update-intensive applications. A transaction can write its log and/or updated data pages to safe RAM to commit. In this way, a response time improvement can always be realized when safe RAM is added to a system, or throughput can be improved to the extent that the system has had to limit disk utilization to achieve adequate response times. Safe RAM also allows group commit to be more fully exploited without hurting response time, even when log files are heavily parallelized. A cost–performance model quantifies when the use of safe RAM is cost-effective.

High-Availability Recovery—In [19], two recovery techniques are examined and compared: mirroring and interleaved declustering. For higher availability, either of these techniques could be used in Bubba in conjunction with a checkpoint-and-log technique using CIR's, each of which provides recovery from a full IR failure. To avoid having to make periodic checkpoint copies from the IR's, the CIR's apply logs to previous checkpoints to obtain new checkpoints. Interleaved declustering provides nearly the same data availability as mirroring, while providing significantly better cost performance. Spare on-line nodes can be used to reduce the time to restore a node and to reduce human operator involvement.

D. The First Prototype

1) Description: By the spring of 1986, progress had been made on a number of important aspects of the design. The overall hardware architecture of the system was defined, an initial design of the FAD language had been proposed, and decisions on the query execution model and transaction management had been made. There was however no clear picture of the software architecture of the system as a whole. Thus, the definition of this architecture, incorporating many of the basic techniques that had been proposed for Bubba, was the goal of the first prototype system.

The system was designed by identifying modules for important system functions, such as the FAD compiler, FAD abstract machine code interpreter, transaction loader, synchronization node (for concurrency control), associative routing manager, FAD object manager, file and index manager, buffer manager, workspace and recovery manager, and so on. The functionality for each of these modules was sketched at a fairly high level, and then external modules interfaces were defined. Each member of the group was given responsibility for the detailed design of one or two modules, which were specified using textual descriptions and an informal pseudocode.

Out of convenience, the prototype software was initially developed on a VAX 11/780 running BSD 4.3 UNIX. The FAD compiler and interpreter were developed using C, LEX, and YACC. The database system code was written in C++, mainly to assess the utility of this new object-oriented language in a large system project. A multitasking environment was provided by a specialized database kernel, called KEV [51]. The KEV interface was designed to be comprehensive yet simple, so that it could be easily implemented on a variety of hardware platforms, for system portability. For this first implementation, KEV's multitasking functionality was emulated within separate UNIX processes using the CSIM process-oriented simulation package, developed at MCC [38]. Many UNIX processes could be used to represent virtual IR's.

To better test and demonstrate parallel execution, the prototype software was later ported to a network of Sun workstations connected by Ethernet. The software ran on diskless workstations; a shared disk server contained the database files corresponding to each virtual IR. In this network environment, a single virtual IR process was run on each of three diskless workstations; a fourth workstation ran user interface (``console'') processes for the three IR's. The configuration is shown in Fig. 4.

The prototype eventually consisted of approximately 30 000 lines of C++. It was able to run multiple order-entry transactions concurrently on three IR's. The system was built as a ``thin vertical slice'' of the Bubba system. That is, the basic parts of all of the major components of the system were implemented but a fair amount of functionality was left out. This allowed all layers of the system to be examined without incurring the time and expense of a complete implementation of the software.

2) Lessons Learned: A number of important lessons were learned from the construction of the first prototype.

Distributed Execution—The prototype contained the first complete, but simple, design of the distributed execution facility of Bubba, including associative routing, a dataflow control method, and transaction management. This initial attempt at distributed execution gave us our first exposure to the costs of parallelism (processes, messages, and delays), and showed us that a single strategy used for program loading, program activation, and dataflow control would not be efficient for all cases. It also prompted us to be a little more frugal with processes in the next design. For example, program loading and transaction commit coordination functions, which had been performed by separate tasks called transaction loader (TL) and synchronization node (SN), were combined into one process which is now the transaction coordinator (TC).

Program Loading and Activation—Program loading and activation in the prototype were done by always sending and executing a component's code at every IR in the home of an operand relation. While this scheme is comprehensive, it frequently yields unnecessary costs for component startup, messages and processing. For example, a ``rifle-shot'' query in which a single tuple is selected will result in one IR in the home finding the tuple, while the other IR's find nothing. We extended the design to allow for dynamic loading and activation, so that components are executed only on the IR's that we know will perform use-

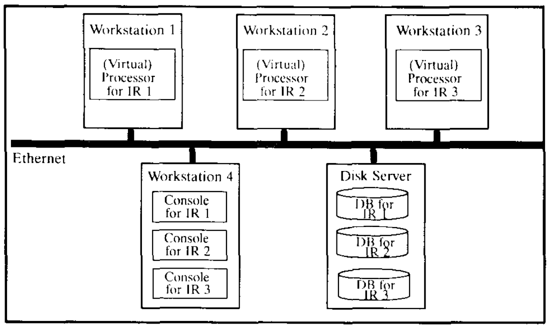


Fig. 4. The first prototype.

ful work. Also, frequently executed programs can now be preloaded and cached at the IR's.

Dataflow Control—Until the first prototype was designed, we did not recognize the need for dataflow control. Even though all IR's in a home executed a component due to the program loading and activation scheme, it was not known until run-time which IR's would actually generate and send dataflow results. Each IR had to inform an elected IR which IR's were sent data, and the lists had to be combined to find out which IR's did not send or receive anything. Null messages had to be sent to IR's that did not receive data. Later, the introduction of dynamic activation resulted in an added complication, in that the number and identity of the sending IR's is not known until run-time. This led to the design of the three flavors of dataflow control with different performance characteristics.

PFAD and Components—The need for an intermediate parallel language such as PFAD was not recognized until the design of the prototype was begun. The development of PFAD provided much insight into the parallel execution of FAD programs. However, the implementation pointed out gaps in our semantics for the execution of a component across multiple IR's. For example, the first prototype supported only one type of send (associative). When considering the execution of a variety of transaction types, we discovered the need for handling the sending of "null" data (to some or all of the threads executing the component) and encountered a number of cases in which other types of sends (e.g., broadcast) are needed. The Bubba design now includes a wide variety of sends [27]. All PFAD programs run on the first prototype were "compiled" by hand. This process taught us about the techniques that would be required to compile FAD programs into a form that could be executed on Bubba.

Object Management—The prototype used three different formats for objects (disk, memory, and message). The need for conversion (and copying) of objects among the different formats resulted in inefficiency and complexity of the object management software. Also, heavy use of object tables added both complexity and inefficiency to the object management functions. As a result, the design of object management in Bubba now emphasizes the uniform object format for transient, persistent, and dataflow objects and the minimization of copy operations for objects [16].

Operating System—Recognizing that database functions require special support not found in conventional operating systems, KEV [51] was designed to provide only a basic set of primitives, leaving as much control as possible to the higher software layers. However, KEV was later replaced by BOS, to include functions that had previously been implemented above the operating system (e.g., buffer management and locking). This change was motivated by two factors: 1) a desire to streamline the software architecture and improve performance through a closer integration of the database system and operating system functions, and 2) the potential to exploit features such as memory-mapped files and lightweight processes (such as in Mach [1]) that are best provided in the OS.

Implementation—We learned a number of lessons about the process of building a complex system. The importance of good source code control procedures in a large development project was made evident by problems due to the lack of such procedures in the early phases of development. The importance of paying attention to "details," such as system bootstrapping and utilities, early on in the project, was also learned the hard way. Bugs in the (slow) C++ translator available to us at the time, the lack of a good C++ debugger, and our lack of fluency in C++ programming caused us to revert to C in the next prototyping phase.

Because of the hardware platform on which the system was built and implementation shortcuts such as the use of simulation software to emulate KEV functionality, the prototype could not be used for meaningful performance experimentation. In any case, this prototype was crucial to providing a context in which important issues could be examined in the detailed design of the software for Bubba.

IV. A More Realistic Prototype

A. Approach

At the end of the first prototyping phase, we were faced with the choice of trying to modify the code of the first prototype or starting from scratch. After much debate and soul-searching we opted to "let go" and start the second prototype from scratch. There were a number of reasons for this decision.

- The desired software architecture and many fundamental algorithms had changed radically since the first prototype so that it was not clear how much (if any) code could be salvaged.
- We wanted to get away from the poor C++ programming tools we had, and instead use familiar C tools.
- The first prototype was coded mostly by part-time students, many of whom were no longer with the project.
- The first prototype had robustness problems that were not easily fixed.

In retrospect, the decision to start from scratch was a good one. We were able to more cleanly implement the new design, interfaces among modules were better defined, and we were able to use techniques such as walkthroughs and code reviews to improve quality and to give the entire group a working knowledge of the major parts of the system. As a result, the system was much more robust than had we ported the earlier code to the new system.

The main goal of the second prototyping effort was to implement the most recent Bubba design, which incorporated the lessons learned from the first prototype, as closely and realistically as possible. This was accomplished with the following exceptions, made for pragmatic reasons.

• We recognized we were building an experimental prototype and not a commercial product. Thus, some important Bubba features were not implemented, such as node recovery, data placement reorganization, and collection of statistics for the FAD optimizer. Also, the prototype was not made "industrial strength" nor has it been "tuned."

• We implemented the system using mid-1980's hardware technology and cost (i.e., the Flex/32 multicomputer), although Bubba was designed for 1990's hardware technology and cost. Thus, some Bubba features were not implemented, such as safe RAM (due to lack of uninterruptible power supply), IP's (due to lack of nodes), and full data and index caching (due to small node memory). While these hardware technology and cost limits impact prototype implementation and performance, they have not resulted in basic design changes.

In order to construct the system in a reasonable amount of time, we had four groups working in parallel: FAD compiler, distributed execution, object management, and BOS. There was frequent coordination among the groups. A phased approach to building the system was used. During the first phase, BOS and the distributed execution software were developed on the parallel machine, while at the same time, the compiler and object management software were developed on Sun workstations using UNIX. During the second phase, the object management software was ported to a single node of the parallel machine and tested in a centralized fashion (no parallelism). During the final phase, we integrated the distributed execution software with the object management so that we could run transactions in parallel. This phased process was quite successful, as it allowed the major development efforts to proceed in parallel with minimal interference, and then integrated and tested incrementally.

B. Hardware Platform

The current Bubba prototype is implemented on a 40-node Flex/32 multicomputer, illustrated in Fig. 5. Each node consists of a 16 MHz Motorola 68020 CPU, 68851 MMU and 68881 FPU, with 2 MB of on-board SRAM. In addition, each node has a set of VME peripherals including 4 MB of external DRAM, a 180 MB 5-1/4" CDC

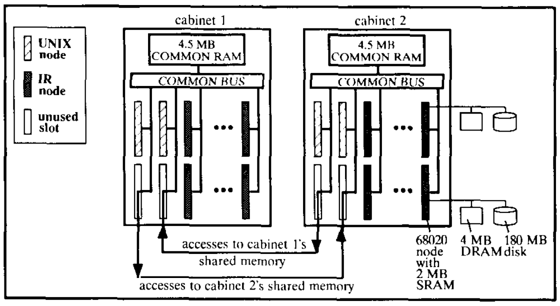


Fig. 5. 40-node Flex/32 hardware platform.

Wren III winchester disk and a high-performance ESDI disk controller. Two cabinets house the 40 nodes.

The per-node sizes of main memory and disk were much smaller than what we expect in a real Bubba implementation. For that reason, we have had to relax some of the Bubba memory management policies in the implementation. Specifically, these are the following.

• System data, cluster indexes, and frequently accessed persistent data may not be cached. Instead, they are subject to regular LRU replacement. (This can be easily changed for experiments requiring caching.)

• The persistent portion of the virtual address space has been limited to 256 MB (out of 4 GB total virtual address space) per node, to keep the page tables smaller. (However, this is sufficient for the 180 MB disk.)

The Flex nodes do not communicate via a hypercube, as we would prefer in Bubba. Instead the nodes share access to 9 MB memory via a hierarchy of 32-bit-wide buses. The bandwidth of the buses and memory in a cabinet is high enough so that there is not a significant bottleneck even when all nodes are accessing the memory at the same time. The system has been configured so nodes can access the common memory in the other cabinet transparently; references are automatically routed across a 32-bit intercabinet bus (there is a bus in each direction). In keeping with our shared-nothing message-passing architecture, the common memory is treated only as a "wire" to hold messages-in-transit between nodes. The common memory is partitioned so that each node has a dedicated buffer pool for outgoing messages, managed exclusively by that node without internode locks or critical sections (hot spots). In this way, our message performance has been made relatively independent of the number of nodes sending and receiving messages simultaneously. Message response times are fairly uniform between any pair of nodes; we did not attempt to simulate the "multihop" delays of a hypercube (although we could have).

Thirty-two of the 40 nodes are used to implement IR's. Rather than sacrifice separate nodes for IP's, we chose instead to run TC processes on IR's so that we would have as many IR's as possible. In each cabinet, two nodes run UNIX and are used for software development and experiment control, and two nodes are unused so that their slots can be used for the incoming and outgoing intercabinet links.

C. FAD Compiler

1) Implementation: The FAD compiler [48] performs static type checking and inference, followed by query optimization with respect to an architectural cost model, followed by parallelization to form a PFAD program of components and arcs, followed by translation to a *load module* of compiled C code and data structures that describe the components and arcs.

Objectives that guided the design of the compiler were: modularity, to allow a number of difficult problems to be addressed separately; and ease of module integration, to allow rapid combination of individual solutions into a working compiler. Modularity was achieved by an architecture consisting of six modules, each with a well-defined role. Ease of integration was achieved by sequencing solutions as individual phases of compilation, and by using a source-to-source language translation approach for each phase. Because intermediate text files are always in FAD (PFAD is simply FAD with components and send/receive operations), visual inspection of module output was possible and testing could be performed module by module. An additional benefit of this approach was the ability to bypass optimization and/or parallelization phases to produce an early version of the compiler targeted for the Sun workstation for demonstration and testing.

The compiler architecture is depicted in Fig. 6. Each crucial compilation issue is handled by a separate compiler phase: type checking by the Analyzer, optimization by the Rewriter, parallelization by the Parallelizer, and low-level code and load module generation by the Translator. The parser for each phase of the compiler is generated from a FAD attribute grammar [2]. The basic FAD grammar remains the same for each phase (the FAD grammar for the Translator is enhanced to include component definitions and send/receive expressions).

The Schema Manager provides uniform access to schema information, hiding implementation details. Three levels of description are provided by the schema: conceptual, internal, and physical [29]. The *conceptual schema* describes the conceptual data (objects and values) assumed by the FAD program and is accessed by the analyzer. The *internal schema* describes how conceptual FAD sets are mapped to internal FAD sets, and is accessed by the optimizer. It shows direct relations [50] and inverted-file relations (secondary indexes) but not declustering. The *physical schema* describes the way the conceptual data are actually implemented in Bubba and is accessed by the Translator. The schema is generated by a separate compiler that processes data descriptions expressed in FAD DDL.

2) Lessons Learned: While implementing the FAD compiler, we learned several lessons, concerning the FAD language as well as the compiler technology.

The FAD compiler automatically infers the types of data objects and expressions in a program when unspecified by the programmer, demonstrating a very effective way to

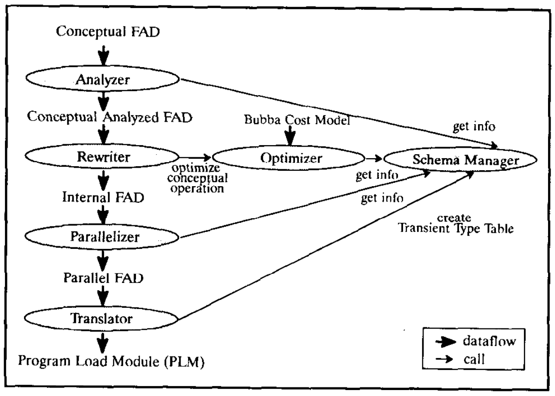


Fig. 6. Compiler architecture.

use available schema information during compilation. Explicitly typed expressions are treated as a form of assertional documentation that is checked by the compiler. However, unification-based type inferencing, well understood in conventional language theory, had to be extended for the database environment of FAD. One example is that a conceptual FAD record type does not specify an ordering of attributes, yet corresponding physical types do. While two conceptual record types with the same attributes may unify, is is also necessary to guarantee that they are ultimately mapped to the same physical type. The FAD compiler uses dataflow analysis to guarantee that a transient tuple that might be inserted in a persistent set is created with the same physical type as that of the set elements, to avoid run-time type checking and conversions.

The common attribute grammar-based framework [45] provided an excellent foundation for parsing in all of the compilation phases, and was especially useful for organizing the unification-based type inference in the Analyzer and abstract program analysis in the Rewriter when accumulating information necessary for optimization. Since the attribute grammar framework best supports program transformations that retain the same basic program structure, it was only of limited use by the Parallelizer, which must generate program transformations that are quite different in structure from the original program.

In the first specification of FAD [9], every data item in a program was an object, each with its own identity [30]. In the second specification of FAD [20], the notion of "values" was introduced. Unlike objects, the notions of identity, sharing, and updating do not apply to values, so they can be implemented much more efficiently by the compiler and DBMS. However, distinguishing between objects and values may be of less general utility than we originally thought. Also, it led to a number of complications in type inferencing. Next time, we would consider supporting only atomic values in addition to atomic and complex (updatable) objects.

Query optimization for a general purpose database programming language like FAD is far more difficult than for a relational query language. In FAD, action expressions can be constructed in an arbitrarily complex fashion. Furthermore, FAD programs can deal with arbitrarily nested objects. Our solution [49] combines abstract program analysis (to recognize optimizable operations) and relational query optimization. The presence of declarative action constructors (such as "filter") in FAD made this approach possible.

Parallelizing general purpose FAD programs also turned out to be much more difficult than decomposing distributed relational queries. The richness of the data model (recursive data structures, disjuncts, and object identity), language (recursion, conditionals, and assignment), and model of execution (dataflow activation, associative routing, replication), and our emphasis on performance required new and enhanced dataflow analysis and program transformation techniques [27]. The Parallelizer optimizes parallelization of common "easy" programs, and does a good job of handling less common "hard" programs.

Support for object sharing by multiple parents involves complex issues that we did not resolve in Bubba. Even in single-node systems, efficient solutions to the management of object sharing do not yet exist. Allowing object sharing in a shared-nothing system, where there is a partitioned storage space and object references can span nodes, is even more difficult. We decided to support the notion of object sharing conceptually in FAD, but restrict its use to transient objects (thereby avoiding the storage management issues in persistent space) and only local parents (thereby avoiding internode references).

The semantics of "null" in a general purpose database programming language with strong static typing are complex. The semantics we ultimately developed for FAD are well founded, consistent, and useful.

Mapping FAD to compiled C was straightforward in theory, but we ran into problems with the resulting legal C expressions being too complex for some C compilers.

D. Distributed Execution

1) Implementation: The distributed execution subsystem was redesigned for the prototype in order to incorporate the more efficient dynamic program loading, activation, and dataflow control protocols. Furthermore, the underlying distributed process model provided by the operating system was redesigned to support multiple concurrent threads per transaction process within an IR.

Transaction programs, components, and arcs are mapped onto the following BOS constructs for execution.

- For each instantiation of a transaction program, a single separate *clan* is allocated in each IR that is participating. The clan provides the virtual address space for all of the threads of that transaction in that IR.
- For each activated component, a separate thread executes the code for the component in the context provided by the clan. In addition, other system threads are used to control the execution of the transaction (such as commit and dataflow control processing). The system threads also execute in clans belonging to the transaction. A thread is the smallest unit of scheduling within a transaction.
- For each incoming dataflow arc for a component, a separate message queue is allocated by each component thread.

In this way, transactions are implemented as parallel processes distributed across the relevant IR's, and facilitate intertransaction, intratransaction, and intracomponent parallelism. Fig. 7 illustrates the multithreaded process structure:

- one *transaction coordinator* (TC) thread communicates with the user, loads and starts the transaction, monitors dataflow execution, and serves as the "master" in the distributed two-phase commit protocol;
- *executor threads* execute PFAD components at an IR;
- zero or more *dataflow control (DFC) threads* coordinate communication between executor threads (Fig. 8 illustrates the dataflow control methods), and inform the TC which nodes are participating in the transaction;
- one *commit thread* per participating IR commits the transaction's updates at the IR under the two-phase commit protocol.

In case of errors or timeouts, the TC coordinates transaction abort and restart (if the error is nonfatal). To support recovery from a TC failure or lost commit message, one or more *log processes* on designated nodes are kept informed of a transaction's progress. Upon failure, the log processes are queried and transactions are recovered according to a presumed abort protocol [34].

The TC, executor, and dataflow control threads at an IP or IR call global directory, symbolic home directory, and dataflow control functions at run-time to associatively route or broadcast results to component threads in other IR's. The global directory and symbolic home directory are cached in memory, since they are consulted often (once for each tuple that is routed). The dataflow control functions prepend a program id to any message that may cause dynamic activation. When a message arrives at an IR for a thread that does not yet exist, BOS will automatically create it. If the program has not been preloaded and cached in the IR's (only done for "bread-and-butter" programs), the TC will send the program code to the IR's that are activated during the course of the dataflow execution.

2) Lessons Learned: Many of the lessons we learned about distributed execution in Bubba came from an extensive experiment to investigate the performance of the dataflow execution strategy upon the prototype [40]. In particular, we were concerned about possible negative effects of parallelism on response time performance, including delays due to asynchronous starting, synchronizing, communicating between and terminating a set of parallel threads. We were also concerned about the effect of contention between concurrent threads from the same or other transactions. We measured the impact of these negative effects using a metric we defined, called response time "skew."

At the time of that experiment, the object management

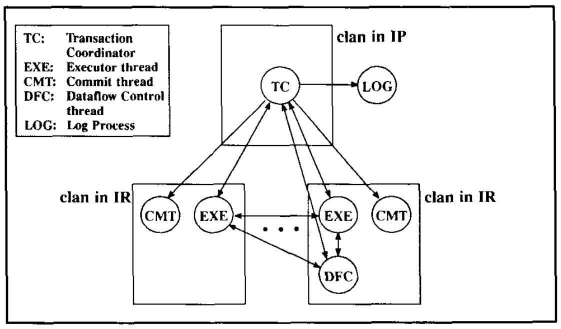


Fig. 7. Process structure for distributed execution in Bubba.

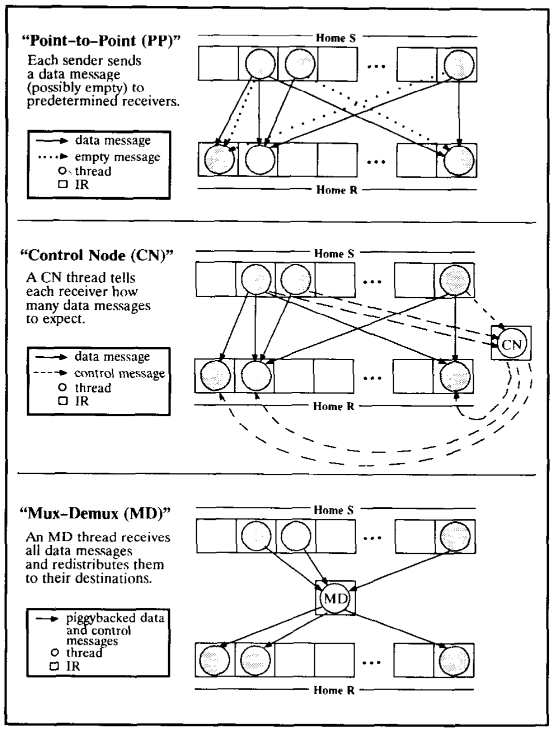

"Point-to-Point (PP)"
Each sender sends a data message (possibly empty) to predetermined receivers.
data message
empty message
thread
IR
Home S
Home R
"Control Node (CN)"
A CN thread tells each receiver how many data messages to expect.
data message
control message
thread
IR
CN
"Mux-Demux (MD)"
An MD thread receives all data messages and redistributes them to their destinations.
piggybacked data and control messages
thread
IR
MD

Fig. 8. Dataflow control methods.

and operating system software was being redesigned, prompted by the lessons of the first prototype. In lieu of the actual object manager and OS, a simulator at each node executed the estimated number of CPU instructions and disk I/O's we expected for that transaction. The distributed execution software was "real," however.

The experiment looked at three different classes of parallel transactions in order-entry: *simple update*, *large scan*, and *large N-M join*. The experiment was run on configurations varying from 4 to 32 nodes. Detailed event traces were captured and analyzed to determine the effect on both throughput and response time.

The results showed that for the class of simple update transactions, throughput scaled linearly and response time is *not* strongly affected by increased parallelism. These transactions had fixed degrees of parallelism. If these types of transactions dominate the system workload, then increasing the degree of declustering can help throughput through intertransaction parallelism, without significant response time penalties. For these workloads, more declustering leads to better performance [15], [44].

For large scan transactions, the results showed response time delays due to start and termination overhead is much less troublesome than expected (at least for degree of declustering ≤ 32). The major aim of declustering, namely load balancing, seems to be more important to good system performance than the relatively small increases in response time due to the temporal skew caused by many asynchronous threads in parallel IR's.

For large *N-M* join transactions, the results showed that performance degrades as declustering increases, which is in agreement with our analytic model [15]. But the problem is the large number of messages required, not process delays. The number of messages per transaction $O(N*M)$ may be reduced in any of three ways.

• The two relations can be nested, transforming the large join into a large scan. However, this is only practical when the relations have a 1-1 or 1-*m* relationship and where this does not seriously degrade the performance of other transactions in the workload. For *n-m* relationships, the relations must be normalized to the extent that data redundancy is eliminated.

• The two relations can both be declustered across the same IR's using the join key, either ahead of time or dynamically. Again, this may be impractical for *n-m* relationships.

• The *degree of declustering*, *N* or *M* can be reduced. However, this may degrade the performance of other transactions.

When these techniques cannot be used, large *N-M* joins will require very efficient communications in a shared-nothing system. Bubba is designed to have fast dedicated message processors (not available in the prototype) and an efficient communications protocol.

For large scans and large joins, methods for parallelizing broadcast messages (e.g., preloading, preactivation, and commit messages) can parallelize the CPU message work of the sending node, and thereby reduce start and termination delays, without increasing the total message work. For example, a scheme was proposed for Gamma [22] to distribute message sequences in parallel down a logical binary-tree imposed on the nodes [25], [26]. As an alternative to associative routing, however, these techniques can improve response time only at a large expense in total message work, because each piece of data is moved several times, and in extra filtering at the receiver.

It is important to limit the multiprogramming level (MPL) at each IR and control the transaction mix. It appears that response time is "well-behaved" for reasonable MPL (i.e., where thrashing is avoided), so that standard scheduling mechanisms should suffice. Since large transactions can *greatly* affect other transaction response times, it would be worthwhile to distinguish and segregate job classes (e.g., using shortest job first scheduling).

A lesson we have yet to learn is how response time would be affected if Bubba's "set-at-a-time" dataflow control protocols were replaced by schemes that allow more pipelining between connected components. On one hand, performance might be expected to improve due to the increased level of concurrency in a transaction. On the other hand, pipelining may increase the variation in response times due to increased levels of asynchronous activity, protocol complexity, and competition for the processor.

E. Object Management

1) Implementation: The object management software of the first prototype was fairly traditional, in that it contained separate layers for FAD object semantics, indexing and record management, file management, and buffer management. The notion of single-level store was a significant "liberator" when redesigning more streamlined object management in the second iteration [16].

The single-level store abstraction allows the same representation to be used for an object whether it is persistent, transient, disk-resident, or memory-resident. That representation is known and easily handled by the compiler, allowing it to build complex constants at compile time, and generate code to directly access objects. Virtual memory pointers are used freely to refer to other objects. Persistent objects are located in a virtual address space that is shared among BOS processes, exists independently of any particular process, and is mapped to disk.

Even with the single-level store, complex objects are still organized into blocks so that objects that are likely to be referenced together can be collocated and retrieved in a single disk I/O upon an access fault. Objects are stored either with or without a cluster index, depending on size. Objects that are larger than a block are provided with indexes which map the object's subobjects (e.g., tuples of a set) into blocks (see Fig. 9). Both *B*-tree and hashed indexes are supported. Since blocks are large, one index block can be used to index very large objects. For example, assuming 16 kbyte blocks and 20 bytes per index entry, one index block can index about 10 Mbytes of data. Assuming 64 kbyte blocks (which we expect in future versions of Bubba), the number increases to about 200 Mbytes. Because of its greatly reduced size, a cluster index can usually be cached. Within each block, a smaller index provides access to objects within the block. This allows any subobject of a large object to be accessed in at most one I/O, because this lower level index is accessed in the same I/O as the data itself.

Garbage collection and clustering of updated persistent objects into blocks are performed by a process called boxing. Boxing can be performed either at the end of a transaction during commit time, or in a background mode.

The use of a single-level store opened up the possibility for MMU-assisted locking of data pages (also exploited in the IBM 801 [12]). We distinguish between two types of pages.

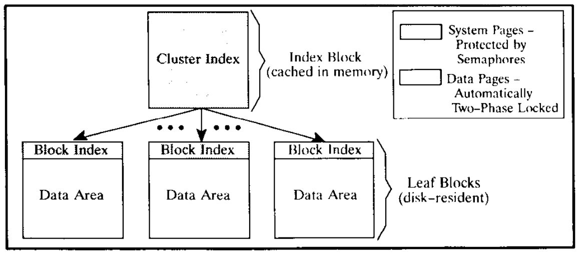

Cluster Index
Index Block (cached in memory)
System Pages - Protected by Semaphores
Data Pages - Automatically Two-Phase Locked
Block Index
Data Area
Block Index
Data Area
Block Index
Data Area
Leaf Blocks (disk-resident)

Fig. 9. A large object and its index.

- *Data pages* are used to hold persistent user data and are implicitly locked. When a memory location in a persistent data page is read (written), the page is automatically read (write) locked. Lock faults occur when a process that accesses a persistent data page does not have the proper lock (read or write) for the attempted access, and are detected by the MMU using its read and write protection bits. When a lock fault is detected, a lock on the data page is obtained if possible. Otherwise, the process requesting the lock is blocked until the lock becomes available. Upon write accesses, a private copy of the original page is made and placed into a differential page-map table for the updating process. This implementation of a "*copy-on-write*" *workspace* was used to simplify transaction aborts. If a transaction aborts (or terminates without committing), its locks are released and any differential pages are discarded without affecting the persistent object space. When a transaction commits, a special commit system call to BOS is used to release all of the transaction's locks and move the process' differential pages into the persistent space.
- *System pages* are used to hold index or control data and are not implicitly locked. Explicit semaphores are used by the index manager to protect data on these pages.

Persistent virtual memory is allocated by BOS in units of blocks. A block can begin with zero or more system pages (specified to BOS when the block is allocated), with the rest of the block composed of data pages (see Fig. 9). By storing indexes and control information in system pages, we are able to avoid contention problems that would result from two-phase locking hot resources. The distinction between system pages and data pages solves one of the main problems in operating system support for locking [41], [43].

Explicit nontwo-phase concurrency control is based on user supplied semaphores. The programmer allocates semaphore structures located in memory that is accessible by all processes that may use it; usually semaphores are collocated with the objects they control. Semaphores are implemented efficiently by $P(\ )$ and $V(\ )$ macros which run in user mode and only call BOS when it is necessary to block or unblock a process. A classic problem with using semaphores that are outside the operating system is that the OS may time-slice a process (or block it for some other reason) while it holds a semaphore, tying up the resource controlled by the semaphore. We address this problem by allowing processes that hold semaphores to indicate to BOS how they should be scheduled.

It is commonly accepted that general-purpose operating system policies often conflict with the atypical needs of database processing [35], [41]. While BOS attempts to implement virtual memory in a way that is useful to the object manager, the object manager often needs to control the paging policy for good performance. Thus, special BOS calls are provided to

- fix pages in memory (during a process); mark blocks as cached (across processes).
- explicitly obtain or release automatic locks.
- cause pages to be read from or flushed to the disk or discarded.

Fixed and cached pages are ignored by the page replacement policy. This is used to keep either temporarily or permanently hot pages (e.g., index pages used often within a transaction or root index pages that are always hot) in memory. When a group of pages needs to be locked, read, or written, it can be done with one system call rather than on a page by page basis, which is important for full or partial scans.

The object manager directly supports associative access to relations through cluster indexes. Inverted files (i.e., secondary relations) are implemented outside the object manager as normal declustered relations. We did not implement inverted files locally for each IR's segment of a declustered relation since this has been shown to require activation all of the IR's containing that relation (e.g., Teradata [23]). The association between conceptual relations and inverted files is indicated in the schema; their use by the compiler's optimizer is transparent to the FAD programmer. An inverted file of a relation R on attribute A, say R_A, is a binary relation whose cluster attribute is A and whose second attribute is one or a set of keys of R. R_A is declustered in the usual way (not necessarily over the same IR's as R), as indicated in the global directory index. There is a local cluster index for R_A within each of its IR's. The FAD compiler generates a separate component for each access to R_A. This approach introduces communication between access to R_A and R, but actually reduces total message and processor work involved in startup and termination for selective transactions and queries (which is expected from a secondary index).

2) Lessons Learned: We have found that the single-level store abstraction can greatly simplify and streamline object management, and promises substantial performance improvements. We recognize that special-purpose OS support is required, but our success in extending UNIX to do it is encouraging (see Section IV-F).

After our initial experience, we are mixed in our assessment of implicit locking. The advantages of this automatic locking mechanism are that 1) it uses standard hardware to efficiently support the often-used function of lock checking, and 2) programs need not be aware of locking or page boundaries. The latter reason is especially important for supporting the more general-purpose (compared to SQL) programming environment of FAD. The disadvantage is that the object manager (or higher levels) has more knowledge about the semantics of operations and, hence, can sometimes use more efficient nontwo-phase locking. Our conclusions are that implicit locking is not a good idea for high-throughput systems that cannot tolerate the data contention involved in strict two-phase locking, unless some persistent data can be exempt from this automatic mechanism. However, implicit locking would be quite beneficial for applications in which data contention is not a bottleneck.

Copy-on-write workspaces simplify aborts by avoiding the need for undo logs, but at the expense by making the actual updates more costly. For example, copy-on-write allocates and copies entire pages even when only a few bytes on a page are updated. Since aborts are infrequent, conventional update-in-place and recovery schemes may lead to better overall performance. We did not fully investigate the potential of copy-on-write for simplifying recovery and logging, or for reducing data contention by allowing readers who could tolerate slightly out-of-date data to not have to wait on writers.

F. BOS

1) Implementation: The current version of BOS [13], [14] was implemented by modifying the Flex/32 version of AT&T UNIX System V Release 2.2. We chose UNIX as a base primarily for expediency. The following extensions had to be made to UNIX to support the BOS functionality used in distributed execution and object management:

Memory Management

- Provide a shared persistent data region used to implement the single-level store.
- Support the two page sizes for memory and disk allocations.
- Perform automatic two-phase locking upon page access faults.
- Implement the copy-on-write differential-page process workspace for updates to persistent data.
- Provide buffer management support, such as page fixing and flushing.
- Support prefetching for full or partial database scans.

Process Management

- Implement multithreaded processes.
- Provide a fast implementation of semaphores which call the operating system only when blocking is necessary.
- Support message-based task control that provides for dynamic loading, activation, and termination of processes and threads.

Messages

- Provide a message interface that includes: multicasting, large numbers of (logical) message queues per thread; short control messages embedded in message headers; and multipage message bodies that are moved with MMU remapping techniques rather than copying.

The implementation of the single-level store in BOS requires the presence of an MMU that supports at least a 32-bit virtual address space, and small (512 byte) pages. The 32-bit virtual address space limits the maximum size of the persistent data space on a single node to less than 4 Gbytes. Currently, BOS supports only a single persistent space. One way to overcome this 4 Gbyte restriction is to provide multiple 4 Gbyte persistent spaces in the context of the UNIX file system namespace, where only one persistent space could be attached at a time. This is similar to segment registers used in processors to extend the address space. Currently, access to the persistent space is unrestricted in the sense that there is no user-based or process-based security. This could also be accomplished by mapping the persistent space to user and group owned UNIX files.

The current implementation of semaphores allows user processes to communicate with the kernel via a shared memory area. This requires a certain amount of trust in the nonkernel code that might not be acceptable on a general purpose operating system.

2) Lessons Learned: Basing BOS on UNIX was a good decision. First, it gave us a relevant framework of processes and virtual memory that we were able to extend to fully support BOS features. Second, it provided us with many standard tools and components that we did not have to develop ourselves (e.g., bootstrap code, a file system, crash analysis tools, etc.). For those tools that we had to develop ourselves (e.g., a parallel BOS debugger—which was an indispensable tool during the distributed software development), we were able to use standard UNIX tools (e.g., sdb) as a base.

The effort involved in modifying UNIX depended on the BOS feature being implemented. The message and task activation components were highly modular and easy to add with only a few changes to the UNIX code. The single-level store support in general was also surprisingly modular. On the other hand, the original version of UNIX supported 2 kbyte pages, and changing this to support 512 byte pages was a major effort. While not as bad, the support for threads also required broad changes in UNIX source code.

V. Recent Performance Experiments

A. Scalability Experiments

Throughout the project, we used performance models and partial system implementations in experiments to examine different ways of reducing the costs of parallelism. Once the most recent prototype was built, we wanted to see the overall performance scalability using the working implementation.

A recent performance study of Tandem's NonStop SQL Release 2 [24] shows the same types of performance scalability that we want and expect for Bubba. We designed a similar experiment to examine scalability in Bubba. Taking into consideration that absolute performance could be improved with tuning, the results continue to suggest that we have been successful in our attempts to manage and exploit parallelism.

1) Scalability Metrics: Three metrics defined in [24] were adopted for our experiments to demonstrate the different forms of performance scalability in the Bubba prototype.

- On-line transaction processing *(OLTP) throughput scale-up* measures throughput of small transactions as the number of IR's and the database size are increased. The performance goal is to increase transaction throughput in proportion to the relative increase in system size, while maintaining the same transaction response times. Throughput scale-up is obtained primarily by increasing intertransaction parallelism, since each transaction has little inherent parallelism.
- *Batch scale-up* measures response times of "batch programs" (in the terminology of [24], e.g., large decision-support queries) as the number of IR's and the database size are increased. The performance goal is to maintain the same query response times by increasing the system size by the same proportion as the increase in database size. Batch scale-up is obtained by increasing intracomponent parallelism so that the amount of work per node is kept constant.
- *Batch speedup* measures improvements of large query response times as more IR's are added to execute the query. The performance goal is to reduce the query response times by the same proportion as the increase in system size. Batch speedup is obtained by increasing intracomponent parallelism so that the amount of work per node is reduced.

2) Workload, Database, and System Configuration: A subset of the order-entry workload was used as the basis for the scalability experiments. We chose the Order-Shipped transaction as the "OLTP" representative, and Suggested-Order as the "batch" representative. Usually, Order-Shipped accesses a variable number of records from three different relations. For the experiment, we modified Order-Shipped to update exactly six tuples, all of which are accessed through primary (foreign) keys. Suggested-Order does a full scan of a relation and performs two arithmetic computations on each tuple. The programs were written in FAD and optimized and parallelized by the FAD compiler.

For the OLTP and batch scale-up experiments, the amount of data per node was kept constant so that the total database size would be proportional to the number of nodes. For the batch speedup experiment, the total database size was fixed and redistributed over an increasing number of nodes. Data values were synthesized in such a way that the declustering of tuples across IR's was nearly uniform. All relations were fully declustered across all IR's.

We measured the performance of configurations ranging in size from a single node to eight nodes. Each IR disk has a database partition large enough to hold over 100 MB of data. For expediency, we only generated the fraction of the order-entry database used by the two transactions (which included the smallest five of eight relations). In the scale-up experiments, each IR holds 16 MB of the database, or 128 MB for 8 IR's. In the speedup experiments, a single 32 MB relation was distributed over the IR's.

We planned on running the experiments on 32 nodes but we discovered a bug in the Flex firmware that causes the interconnection hardware to fail when running UNIX (BOS) nodes without console terminals. This was a surprise: our earlier experiments [40] had not exposed the problem because UNIX was not used, and later, all of our testing of BOS happened to be on nodes with terminals. As of this writing, we have not been able to solve the problem and we are thus limited to eight nodes with consoles for these experiments. While the small number of nodes is unsatisfying, the preliminary results demonstrate the scalability trends we expected.

3) OLTP Throughput Scale-up: To measure throughput scale-up, we ran workloads consisting of order-shipped transactions against prototype configurations of increasing size. A driver program was set up to maintain a specified degree of multiprogramming (i.e., a specified number of active transactions) in the system. The degree of multiprogramming was increased until maximum throughput was reached, to the point where adding more transactions only lengthened response times without increasing throughput. Fig. 10 plots transaction throughputs for the different configurations, in which a cap of 2 s was maintained on average response time.

Fig. 10 shows that throughput scales-up in proportion to the number of IR's. The slight downturn in the curve from 1 to 4 IR's most likely shows the increasing overhead of parallelism: as parallelism is increased, so are the numbers of clans, threads, messages, etc. (see Section IV-D). However, OLTP transactions have a limited degree of parallelism; Order-Shipped has an inherent degree of parallelism of 6. At that point, IR's can continue to be added without increasing the overhead of each Order-Shipped transaction. The added IR's then only provide more throughput capacity, as is suggested by the upturn in the curve from 4 to 8 IR's. For that reason, we expect the upturn to continue for large configurations.

4) Batch Scale-up: To measure batch scale-up, we ran workloads consisting of single Suggested-Order scan queries against prototype configurations of increasing size. Fig. 11 plots query response times for each configuration size. Fig. 11 shows that response times are about the same for each configuration, even though the database increases in size.

The jump in response time between one to two IR's was expected in Fig. 11. Although the query had been parallelized, when it is run on one IR all the components execute in the same clan, and large intermediate results are sent between components via local message sends (which are implemented using page remapping) instead of remote sends (which copy data pages). We included the single IR results simply as a point of reference.

As the number of IR's increased beyond two IR's, the response time decreases slightly. This is because the database values were generated in such a way that the query would always generate the same sized (and rather large) result. This was done with the intention of maintaining a fixed response time for the final part of the query which receives and prints the result, and cannot be parallelized. The consequence, however, is that the cost of building the result tuples during the scan is spread over more IR's, resulting in a speedup of that part of the query. The effect of this speedup is expected to diminish as the number of IR's is increased.

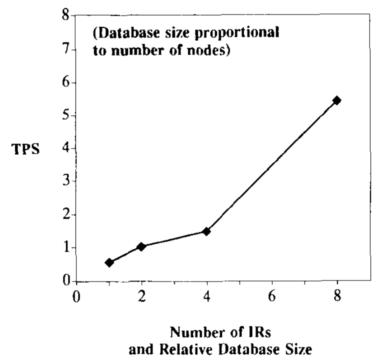


Fig. 10. Throughput scale-up of order-entry OLTP workload.

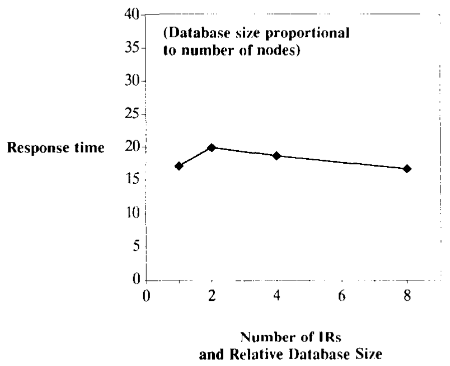


Fig. 11. Batch scale-up of order-entry decision-support workload.

5) Batch Speedup: To measure batch speedup, we ran workloads consisting of single Suggested-Order scan queries against a fixed size database spread over an increasing number of IR's. Fig. 12 plots query response times for each configuration size. The speedup curve in Fig. 13 plots the relative improvements in the response times and shows that the improvements are proportional to the size of the system. We are somewhat surprised by the superlinear increase in speedup between four and eight nodes; it may be due to reduced paging in the IR's because of less data per IR.

The curve in Fig. 12 shows the diminishing returns of "linear speedup." That is, at some point, doubling the system size in order to halve query response time becomes cost-ineffective. That point is determined by the size of

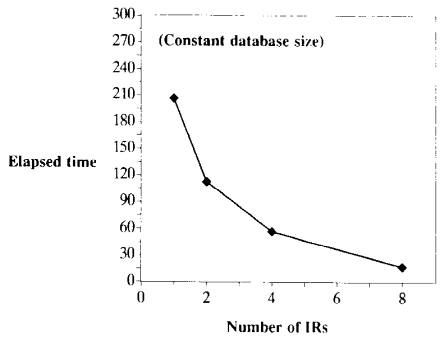


Fig. 12. Response times of order entry decision-support workload.

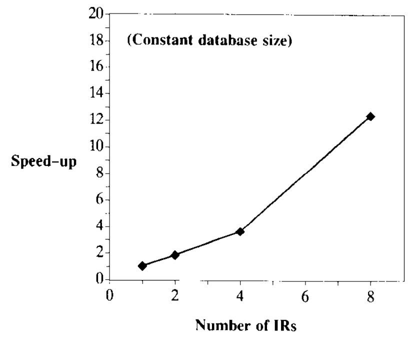


Fig. 13. Batch speedup of order-entry decision-support workload.

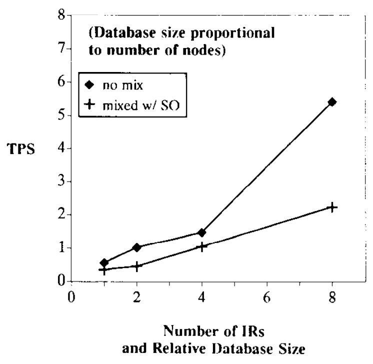


Fig. 14. Throughput scale-up of OLTP transactions in mix.

the original query and the response time and cost requirements of the application.

B. Mixed Workload Experiment

Bubba was designed to support on-line processing of a large database which can be continuously updated. In a second small experiment, we examined the prototype's efficiency in executing a mix of concurrent OLTP and decision-support transactions. We used the same database and workload as for the OLTP and batch scale-up experiments described in Section V-A. The Order-Shipped OLTP transactions do most of their update work on the same item-inventory relation that the Suggested-Order decision-support transactions scan, giving us the data contention we desired in the workload. Suggested-Order scans the item-inventory relation using a nontwo-phase locking option (similar to "browse access"), in order to avoid locking the relation fragments and blocking Order-Shipped transactions for extended periods of time. In this first prototype, this option was implemented by releasing read locks that were acquired for a data block as soon as the next block is scanned.

The mix consisted of multiple Order-Shipped transactions run concurrently with a continuous serial stream of Suggested-Order queries (i.e., there was only one Suggested-Order in progress at a time). The number of concurrent Order-Shipped transactions was increased until the Order-Shipped throughput reached a maximum, while maintaining a 5 s cap on the average Order-Shipped response time. The 2 s response time cap used in the earlier no-mix order-shipped experiments could not be used with the mixed workload, because of the impact of the Suggested-Order load. A Suggested-Order query is both I/O and CPU intensive: each IR executing the query accesses about 8000 data pages and performs two large computations on each of about 1000 selected tuples. This added load has a significant effect on the response time of the relatively small Order-Shipped transactions, due to larger multiprogramming interference and larger "skew" in the start and stop times of a set of threads [40]. Both types of transactions were run using the same starting priorities and scheduling policies; it might have helped to give Order-Shipped transactions higher priority (this is not yet possible in the prototype). Adjusting the size of the scheduler's time slice also would have helped, since Suggested-Order tends to use most or all of each time slice it is granted. (A rather large 1/2 s time slice had been used for this experiment.)

Fig. 14 shows that throughput for Order-Shipped OLTP transactions in the mix again scales up in proportion to the number of IR's and the database size. The slope of the throughput curve is somewhat lower than the no-mix throughput curve because a significant fraction of the system resources are used to execute the concurrent Suggested-Order workload. The total work for the Suggested-Order workload scales with the size of the system and the database, so that the Suggested-Order "overhead" per IR remains constant for any configuration. Thus, as in the no-mix case, the throughput curve is expected to increase linearly for larger configurations, albeit at a lower rate.

Fig. 15 shows that Suggested-Order "batch" response times in the mix increase in proportion to the system and database size, instead of remaining approximately constant as in the no-mix case. This is because Suggested-Order queries were run serially without contention for system resources by other transactions or queries. In the mixed case, each Suggested-Order query had to contend with multiple Order-Shipped transactions, and the number of concurrent Order-Shipped transactions in the system increased in proportion to system size.

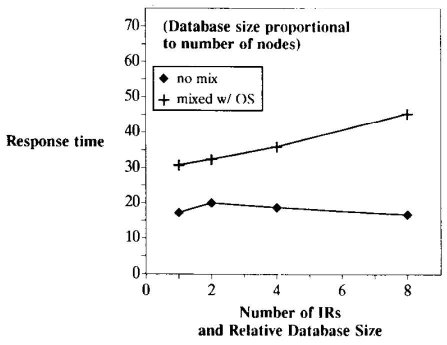


Fig. 15. Batch scale-up of decision-support transactions in mix.

VI. Conclusions

The final Bubba prototype design documentation and software was packaged and distributed to our shareholders in April, and updated in October 1989. The design documentation comprised over 1200 pages of text. There are over 60 000 lines of C code for the FAD compiler and database system software, and approximately 38 000 lines of C code in BOS (of which 9000 were added or changed from UNIX).

We consider these to be the most significant technical and procedural lessons learned during the Bubba project:

Technical Lessons:

• *Shared-nothing is a good idea, but has limitations.* For high-end systems using SQL (requiring >400 TPS), it appears to be the only alternative. For large systems (200–400 TPS), it is much more cost-effective than mainframes. For small to medium systems (<200 TPS), there are many alternatives with similar cost–performance; however, shared-nothing is the only architecture that can scale throughout the entire range. As we discussed earlier, large joins may have trouble scaling well in shared-nothing systems because of the $O(N*M)$ number of messages needed to redistribute the joined relations (in general each of N nodes routes tuples to M joining nodes), especially as N and M become large—the degree of impact will depend on the relative cost of communications versus other join processing. In spite of scaling limitations, overall performance for large joins is likely to be better in shared-nothing than other architectures.

• *Dataflow seems better than remote procedure call (RPC) for a shared-nothing architecture.* A dataflow execution strategy usually reduces the amount of data that must be communicated and allows more parallelism. Even when nonblocking RPC is used, a single node is often a response-time bottleneck, caused by either sending requests to or receiving results from many nodes. When multiple parallel operations are involved, RPC can cause data to unnecessarily pass back through a single node, whereas dataflow allows the distributed result of one parallel operation to be sent in parallel directly to the second distributed operation. Furthermore, the RPC execution model is synchronous, usually precluding execution of multiple program threads in parallel. The RPC-style dataflow is efficient in some situations, however; in fact, one of our three dataflow control methods (Mux-Demux) has simliar performance characteristics. In a distributed (esp. heterogeneous) DBMS environment, however, RPC seems to have the advantage of autonomy and simplicity. Only the single RPC node needs a global directory, and sophisticated dataflow control protocols would not need to be standardized.

• *More compilation and less run-time interpretation seems to be a good idea.* Database people tend to want to do things at run-time, because of their typically strong systems and weak compiler background. We found that many things were better done in the compiler, improving performance because of the leverage in compiling once and running many times and because of reduced lock-holding time. At various times throughout the project, however, it was difficult for some database and compiler people to communicate effectively.

• The *uniform object management concept* (including single-level store and uniform formats) *did simplify the design*. However, in our early view of single-level store, it was bundled with automatic locking and workspaces (which have questionable value for many applications), together providing simplicity and transparency for general-purpose programming. Later, we realized that these were quite separate. Single-level store and optional automatic locking have been considered for use in a more general-purpose standard systems platform, using C and UNIX as a base [17].

• Many systems take the approach of building a node that is itself fault tolerant. In Bubba, we were able to *base our recovery mechanisms on the assumption that an entire node is a field-replaceable unit.* That is, if any part of a node has a hard failure, then the whole node is assumed bad and its data are copied from other nodes to the online spare that replaces it. This scheme becomes less practical if 1) the cost of a node is much larger than the cost of the component that failed, or 2) the database per node is so large that its copy time increases the window of vulnerability for the second copy to the point that availability becomes unacceptable. Our approach seems quite reasonable for smaller nodes (e.g., tens of MIPS and a few disks). Its main advantage is simplicity of design and human operations, both of which are crucial to reliability and availability.

Procedural Lessons:

• *The iterative design-for-performance approach was critical.* Unlike most engineering disciplines, performance modeling is not closely integrated into software engineering; instead, it is generally regarded as a specialized field on its own. Several such modeling specialists were included early in the Bubba project. A considerable amount of time was spent on cross learning between modeling and database people. The modeling people strongly encouraged using a gedanken workload and setting specific performance goals. These gave us something concrete to resolve tradeoffs that otherwise might have left us floundering. Through progressively detailed modeling, many design concepts were either verified so that they became accepted by the group, or were found inadequate early enough to be corrected.

• Although *good software engineering practice* (e.g., design and code walkthroughs, source code version management and periodic massive documentation) are accepted by many, it is difficult to make happen in a research environment. It consumes large resources in both people and time. Nevertheless, it is a must for prototyping and everyone has to simply "bite the bullet" and make the economic and psychological commitment. We found it was crucial for a project of our size.

• *We are glad we did a first prototype.* It provided much insight into how to structure the IR software architecture and made clear to everyone the importance of practicing good software engineering. *We are equally glad we threw it away*, having learned from our mistakes, although it was difficult for most of us to "let go" of it.

• *The idea of a distributed system implementation with single-node simulation was very useful.* We learned much about the overall performance of Bubba much earlier than had we waited for implementation of the single-node software. If the performance results of our experiments had been negative, it would have provided a point at which to either redesign or terminate the project. It also facilitated a phased development of the full prototype implementation, and early development of instrumentation and data analysis software.

• *In retrospect, a good commerically-available hardware platform for prototyping Bubba was hard to find.* We had many problems with the Flex system. The development environment was inadequate and we were plagued with many software and hardware bugs. However, the Flex had the hardware configurability that we needed and was the best choice at the time the decision had to be made.

• *Our project goals were far too ambitious and covered far too much functionality.* This was partially due to our own ambition, and partially due to Bubba being part of the larger ADBS project. It would probably have been better to limit the project to investigating large-scale parallelization of SQL at first. This could have been done more completely and sooner.

Acknowledgment

Over the last five years the Bubba project has had many important contributors, beyond those listed as authors. Much of the final prototype was designed and implemented by J. Bowen, T. Briggs, M. Cochinwala, N. Nowotny, G. Weikum, J. Criollo, S. Krishnamurthi, Y. Hung, and B. Kerola. T. Keller, E. Boughter, C. Buckalew, P. Jenq, B. Twichell, C.-R. Young, R. Brice, and H. Schwetman provided the critical performance analyses, simulations, measurements, and advice that guided the Bubba design process. S. Khoshafian, F. Bancilhon, R. Krishnamurthy, S. Redfield, and K. Wilkinson deserve special recognition for their lasting contributions to Bubba's design during its formative years. M. Holbrook, B. Boettcher, and L. Crider capably crafted the voluminous design documentation. D. Frank and J. Haritsa contributed to the design as summer research students. A. Budinszky, J. Crandell, T. Jagodits, and K. Soheili implemented much of the first prototype. We thank MCC management (in particular E. Lowenthal), our support staff and shareholders for creating the environment in which this work was done. The authors thank the referees and R. Brice for their help in improving the presentation in this paper.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Proc. Summer USENIX Conf.*, July 1986.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[3] W. Alexander and G. Copeland, "Process and dataflow control in distributed data-intensive systems," in *Proc. 1988 ACM SIGMOD Conf. Management Data*, Chicago, IL, May 1988.

[4] W. Alexander and G. Copeland, "Comparison of dataflow control techniques in distributed data-intensive systems," in *Proc. 1988 ACM SIGMETRICS Conf. Measurement Modeling Comput. Syst.*, Santa Fe, NM, May 1988.

[5] W. Alexander, T. Keller, and E. Boughter, "A workload characterization pipeline for models of parallel systems," in *Proc. 1987 ACM SIGMETRICS Conf.*, May 1987.

[6] W. Alexander and D. Kim, "Dynamic vs. static routing in hypercubes," MCC Tech. Rep. ACA-ST-146-88, Apr. 1988.

[7] Anon *et al.*, "A measure of transaction processing power," *Datamation*, vol. 31, no. 7, Apr. 1985.

[8] M. Atkinson and O. Buneman, "Types and persistence in database programming languages," *ACM Comput. Surveys*, vol. 19, no. 2, June 1987.

[9] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, A powerful and simple database language," in *Proc. Conf. Very Large Data Bases*, Brighton, England, 1987.

[10] H. Boral, "Parallelism in Bubba," in *Proc. Int. Symp. Databases in Parallel Distributed Syst.*, Austin, TX, Dec. 1988.

[11] E. Boughter, W. Alexander, and T. Keller, "A tool for performance-driven design of parallel systems," in *Proc. 4th Int. Conf. Modeling Techniques Tools Comput. Perform. Eval.*, Palma, Spain, Sept. 1988.

[12] A. Chang and M. Mergen, "801 Storage: Architecture and programming," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, Feb. 1988.

[13] L. Clay, G. Copeland, and M. Franklin, "Operating system support for an advanced database system," MCC Tech. Rep. ACA-ST-140-89, Mar. 1989.

[14] ——, "UNIX extensions for high-performance transaction processing," in *Proc. Workshop UNIX Transaction Processing*, USENIX, Pittsburgh, PA, May 1989.

[15] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in *Proc. 1988 ACM SIGMOD Conf.*, Chicago, IL, May 1988.

[16] G. Copeland, M. Franklin, and G. Weikum, "Uniform object management," in *Proc. Int. Conf. Extending Database Tech.*, Venice, Mar. 1990.

[17] G. Copeland, S. Danforth, M. Franklin, M. Smith, and L. Clay, "Cworld: Extending the C environment for transaction processing," MCC Tech. Memo—Cworld Memo 5, May 1989.

[18] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith, "The case for safe RAM," in *Proc. 15th Conf. Very Large Data Bases*, Amsterdam, Aug. 1989.

[19] G. Copeland and T. Keller, "A comparison of high-availability media recovery techniques," in *Proc. 1989 ACM SIGMOD Conf.*, Portland, May 1989.

[20] S. Danforth, S. Khoshafian, and P. Valduriez, "FAD, A database programming language," MCC Tech. Rep. DB-151-85, Rev. 3, Jan. 1989; *IEEE Trans. Knowledge Data Eng.*, to be published.

[21] D. DeWitt and R. Gerber, "Multiprocessor hash-based join algorithms," in *Proc. 11th Conf. Very Large Data Bases*, Stockholm, Sweden, Aug. 1985.

[22] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna, "GAMMA—A high performance dataflow database

machine,'' in *Proc. 12th Conf. Very Large Data Bases*, Tokyo, Japan, Aug. 1986.

[23] D. DeWitt, H. Boral, and M. Smith, ''A single-user performance evaluation of the Teradata database machine,'' in *Proc. 2nd Int. Workshop High Performance Transaction Syst.*, Asilomar, CA, Sept. 1987.

[24] S. Englert, J. Gray, T. Kocher, and P. Shah, ''A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases,'' Tandem Tech. Rep. 89.4, Part .27469, May 1989.

[25] R. Gerber, ''Dataflow query processing using multiprocessor hash-partitioned algorithms,'' Ph.D. dissertation, Comput. Sci. Tech. Rep. 672, Univ. of Wisconsin-Madison, Oct. 1986.

[26] R. Gerber and D. DeWitt, ''The impact of hardware and software alternatives on the performance of the Gamma database machine,'' Comput. Sci. Tech. Rep. 708, Univ. of Wisconsin-Madison, July 1987.

[27] B. Hart, S. Danforth, and P. Valduriez, ''Parallelizing FAD, A database programming language,'' in *Proc. Int. Symp. Databases in Parallel Distributed Syst.*, Austin, TX, Dec. 1988.

[28] B. P. Jenq, B. Twichell, and T. Keller, ''Locking performance in a shared nothing parallel database machine,'' in *Proc. 5th Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 1989; also *IEEE Trans. Knowledge Data Eng.*, vol. 1, no. 4, Dec. 1989.

[29] S. Khoshafian and T. Briggs, ''Schema design and mapping strategies for persistent object models,'' *Inform. Software Technol.*, Dec. 1988.

[30] S. Khoshafian and G. Copeland, ''Object identity,'' in *Proc. 1st Int. Workshop Object Oriented Programming Syst., Languages, Appl.*, Portland, Oct. 1986.

[31] S. Khoshafian and P. Valduriez, ''Parallel execution strategies for declustered databases,'' in *Proc. 5th Int. Workshop Database Machines*, Karuizawa, Japan, Oct. 1987.

[32] —, ''A parallel container model for data intensive applications,'' in *Proc. 6th Int. Workshop Database Machines*, Deauville, France, June 1989.

[33] M. Livny, S. Khoshafian, and H. Boral, ''Multi-disk management algorithms,'' in *Proc. 1987 ACM SIGMETRICS Conf.*, May 1987.

[34] C. Mohan, B. Lindsay, and R. Obermarck, ''Transaction management in the R* distributed database management system,'' *ACM Trans. Database Syst.*, vol. 11, no. 4, Dec. 1986.

[35] E. Moss, ''Getting the operating system out of the way,'' *Database Eng.*, Sept. 1986.

[36] S. Naqvi and S. Tsur, ''A logic language for data and knowledge bases,'' MCC Tech. Rep. ACA-ST-176-88, Aug. 1988.

[37] D. Ries and R. Epstein, ''Evaluation of distribution criteria for distributed database systems,'' UCB/ERL Tech. Rep. M78/22, Univ. of California, Berkeley, May 1978.

[38] H. Schwetman, ''Using CSIM to model complex systems,'' in *Proc. 1988 Winter Simulation Conf.*, San Diego, CA, Dec. 1988.

[39] K. Sevcik, ''Data base system performance prediction using an analytical model,'' in *Proc. 7th Int. Conf. Very Large Data Bases*, France, Sept. 1981.

[40] M. Smith, W. Alexander, H. Boral, G. Copeland, T. Keller, H. Schwetman, and C.-R. Young, ''An experiment on response time scalability in Bubba,'' in *Proc. 6th Int. Workshop Database Machines*, Deauville, France, June 1989.

[41] M. Stonebraker, ''Virtual memory transaction management,'' *ACM Oper. Syst. Rev.*, vol. 18, no. 2, Apr. 1984.

[42] —, ''The case for shared nothing,'' *Database Eng.*, vol. 9, no. 1, 1986.

[43] M. Stonebraker, D. DuBourdieux, and W. Edwards, ''Problems in supporting data base transactions in an operating systems transaction manager,'' *ACM Oper. Syst. Rev.*, vol. 19, no. 1, Jan. 1985.

[44] The Tandem Performance Group, ''A benchmark of NonStop SQL on the Debit Credit transaction,'' in *Proc. 1988 ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[45] M. Tiemann, ''ICC: An incremental compiler compiler based on attribute evaluation,'' MCC Tech. Rep. PP-412-86, Dec. 1986.

[46] S. Tsur and C. Zaniolo, ''Logic data language (LDL),'' in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986.

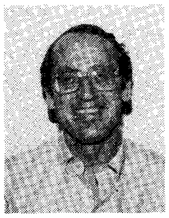
[47] P. Valduriez, ''Join indices,'' *ACM Trans. Database Syst.*, vol. 12, no. 2, June 1987.

[48] P. Valduriez, S. Danforth, B. Hart, T. Briggs, and M. Cochinwala, ''Compiling FAD, A database programming language,'' in *Proc. Int. Workshop Database Programming Languages*, Portland, June 1989.

[49] P. Valduriez and S. Danforth, ''Query optimization in database programming languages,'' in *Proc. Int. Conf. Deductive Object-Oriented Databases*, Kyoto, Japan, Dec. 1989.

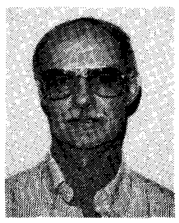
[50] P. Valduriez, S. Khoshafian, and G. Copeland, ''Storage models for complex objects,'' in *Proc. 12th Int. Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986.

[51] K. Wilkinson and H. Boral, ''KEV-A kernel for Bubba,'' in *Proc. 5th Int. Workshop Database Machines*, Karuizawa, Japan, Oct. 1987.



Haran Boral received the B.Sc. degree in computer science from CCNY in 1977 and the Ph.D. degree, also in computer science, from the University of Wisconsin-Madison in 1981.

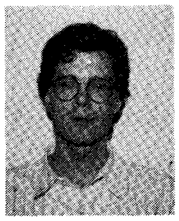
In 1984, after a couple of brief academic appointments, he joined the Microelectronics and Computer Technology Corporation (MCC), Austin, TX, as a member of the Bubba project. After a short period as a technical contributor, he assumed management responsibilities for the project. He also initiated research activities in optical computing and neural networks at MCC. He is now blissfully unemployed.



William Alexander received the B.A. degree from Rice University, Houston, TX, and the M.A. and Ph.D. degrees from The University of Texas at Austin in computer sciences.

He is currently a Senior Member of the Technical Staff in the Advanced Computer Technology Program at Microelectronics and Computer Technology Corporation, Austin, TX. His research interests include performance measurement and modeling and distributed systems.

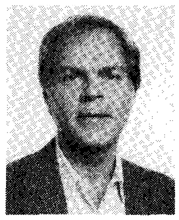
Dr. Alexander is a member of the Association for Computing Machinery.



Larry Clay (M'83) received the B.S. degree in mathematics, the B.A. degree in computer science, and the M.A. degree in computer science from The University of Texas at Austin.

He currently works for Tandem Computers, Austin, TX. He was a member of the Technical Staff at MCC for three years, working on the Bubba project. His research interests include operating systems and performance analysis.

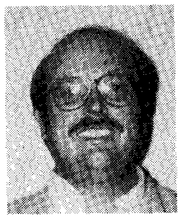
Mr. Clay is a member of the Association for Computing Machinery.



George Copeland (S'66–M'74) received the B.S. degree from Christian Brothers College, Memphis, TN, in 1969, and the M.E. and Ph.D. degrees from the University of Florida, Gainesville, in 1970 and 1974, all in electrical engineering.

He has worked for NASA, Bank of America, and Tektronix, and was a founder of Servio Logic Corporation. AT MCC, he served as chief architect of the Bubba project. He currently works for IBM, Austin, TX. His major areas of interest are scalable and highly-available database and transaction-processing systems, and persistent object-oriented systems.

Dr. Copeland is a member of the Association for Computing Machinery.



Scott Danforth received the Ph.D. degree in computer science from the University of North Carolina, Chapel Hill, in 1983.

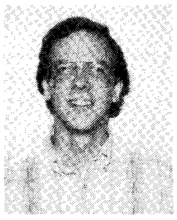
He has been a member of the Technical Staff of the Advanced Computer Technology Program of MCC since February 1984. At UNC, he assisted Gyula Mago in the design of a cellular multiprocessor tailored for direct execution of functional languages. His interest areas include alternative programming paradigms, their associated logics, compilers, and parallel execution models.

Michael Franklin received the B.S. degree in computer and information science from the University of Massachusetts, Amherst, in 1983, and the M.S.E. degree in software engineering from the Wang Institute of Graduate Studies in 1986.

He is currently a Ph.D. degree student in the Department of Computer Sciences at the University of Wisconsin–Madison, and is a Research Assistant on the EXODUS database project. From 1986 to 1989 he was a member of the Bubba project at MCC. His research interests include object-oriented systems, persistent programming languages, and parallel database systems.

Brian Hart received degrees in mathematics and computer sciences from the University of Texas, Austin.

He is currently a research engineer at BULL in France where he works on a parallel database system project. He has been active in the computer industry since 1975, and recently worked at MCC in the Bubba project for three years designing and implementing the FAD parallelizer and several other modules.



Marc Smith received the B.S. and M.S. degrees in computer science from the University of Minnesota, Minneapolis.

From 1980 to 1984, he was a researcher at the Honeywell Computer Science Center, Minneapolis, MN. In 1984, he joined the MCC Database Program in Austin, TX as a member of the Bubba project. He currently works for IBM, Austin, TX. His research interests include parallel database systems, transaction processing, recovery techniques, and performance analysis. He is a member of the Association for Computing Machinery.

Patrick Valduriez received the Ph.D. degree in computer science from the University of Paris in 1981.

He is currently a Director of Research at INRIA, Rocquencourt, France, where he heads a group of scientists working on an advanced database system project. Previously, he spent five years at MCC in the Bubba project where he headed the FAD compiler team. He has authored or coauthored over 40 technical papers and several books on various aspects of database systems.