VINO: An Integrated Platform for Operating System and Database Research

Christopher Small and Margo Seltzer
Harvard University
Cambridge, MA 02138
{chris,margo}@das.harvard.edu

Harvard Computer Science Laboratory Technical Report TR-30-94

Abstract

In 1981, Stonebraker wrote:

> Operating system services in many existing systems are either too slow or inappropriate. Current DBMSs usually provide their own and make little or no use of those offered by the operating system. [STON81]

The standard operating system model has changed little since that time, and we believe that, at its core, it is the *wrong* model for DBMS and other resource-intensive applications. The standard model is inflexible, uncooperative, and irregular in its treatment of resources.

We describe the design of a new system, the VINO kernel, which addresses the limitations of standard operating systems. It focuses on three key ideas:

- Applications direct policy.
- Kernel mechanisms are reusable by applications.
- All resources share a common extensible interface.

VINO's power and flexibility make it an ideal platform for the design and implementation of traditional and modern database management systems.

1 Introduction

In general, operating systems are designed for the least common denominator application. This has proven to be a poor match for resource-intensive applications, such as database management systems. The result is that DBMS avoid using operating system services, and reimplement them from scratch.

Stonebraker points out several ways in which the specific requirements of relational databases are poorly served by traditional operating system architectures in the areas of buffer pool management, file storage, process scheduling, and concurrency control.

For example, the disk buffer cache management strategy most commonly used by operating systems is LRU; this is suboptimal for most relational database disk access patterns [CHOU85].

DBMS have usually dealt with the architectural mismatch by evasion; they overcome operating system limitations by reimplementing and avoiding the use of operating system services wherever possible.

For example, rather than use the supplied filesystem, a DBMS will write directly to the raw disk partition [ORA89, SYB90]. The layout of the data on the partition and the caching of that data then falls under the control of the DBMS. However, because the virtual memory of the database process is still managed by the operating system, in-memory copies of disk buffers are "double-buffered"; they are stored by the DBMS (in memory or on the database partition), and also by the operating system in the backing store for process. In addition, standard operating systems tools (e.g. tools that display the contents of a directory) can not be used. From the operating system perspective, this is akin to the management of "BLOB"s in relational databases; a BLOB is stored by the DBMS, but is not a first-class entity that can be manipulated using the standard DBMS interface. Furthermore, the allocation between the filesystem and the DBMS is completely static; the device containing the filesystem must be taken off-line and rebuilt in order to change the size of a partition.

Transaction support is rarely provided by general-purpose operating systems; its availability at the user level is the exception rather than the rule (e.g. Quicksilver [HASK88]). Instead, operating systems implement one or more special purpose mechanisms to support recovery. For example, most systems support some type of file system recovery service, but this service is rarely exported to the user level, meaning that a DBMS must

implement its own recovery system. Recovery code is notoriously complex, and is often the subsystem responsible for the largest number of system failures [SULL91]. Supporting multiple recovery paradigms is likely to reduce total system robustness.

Object databases are presented with greater barriers to development and performance. Object stores that handle object faulting through the use of virtual memory management hardware support (e.g. [LAMB91]) are faced with the performance impact of a protection fault on first access, and, if the first fault was on a read access, a second on write access. An object database management system that is implemented using a client-server architecture can end up triple-paging; on the server, data is present in both the database and the server's swap file, and on the client it is present both in the client cache or the client swap file.

At times, a service provided by the operating system is frustratingly close to what is needed by a DBMS. For example, Skarra examined the feasability of using standard Posix locking services in a DBMS [SKAR94]. She found that there were several places where commercially available implementations did not correctly detect deadlock, that starvation of write lock requests can occur, that the semantics of lock upgrades is not consistent across different implementations, and that performance is inadequate for DBMS use.

The fundamental problem is that the services provided by the OS are not the services desired or needed by the DBMS.

We next discuss related work, and then the motivation for VINO. In section 4, we describe the system architecture. In section 5 we describe specific VINO subsystems that are of use to a DBMS implementor. We conclude with sketches of VINO-based implementations of relational, object-relational, and object-oriented DBMS.

2 Related Work

DBMS and operating systems have confronted the problems that VINO addresses. Some DBMS implement a unified resource interface and extensibility; some operating systems also offer limited application control of policy.

2.1 Database Perspective

Relational database management systems separate the abstract interface for a tuple from its representation (normally, as an element in a B-tree or hash table). This data independence allows high-level tools, such as query processors, to work independent of the implementation (the *internal level* in ANSI/SPARC terminology [DION78]).

In a relational system, queries are expressed in terms of tuples and keys, without regard for the underlying implementation or indexing structures. The query processor may take advantage of known strengths or weaknesses of various indexing methods, but it accesses the underlying relations through a simple, tuple-oriented interface using calls to retrieve tuples by key, scan sequentially through a relation, or modify specific tuples. It is this abstract interface that we use as the foundation for VINO's resource interface. The system can take advantage of information about the underlying representation, but can function correctly at the abstract level.

The Illustra DBMS [ILLU94] is a successor to the POSTGRES system developed at the University of California, Berkeley [STON87]. It can be extended by adding user-written code to implement new data types. This code can be run in either the client process or the server process; if it is installed in the server, there is no guarantee that it will not corrupt the server or violate system security.

The Thor system, developed at MIT [LISK93], also allows the server to be extended by adding client code. To ensure that client code does not compromise the server, extensions must be written in a system specific typesafe language, *Theta* [MYERS93].

2.2 Operating System Perspective

2.2.1 Extensibility

- *Mach:* Newer operating systems have adopted the *microkernel* architecture [ACET86], which allows the system to be extended by adding user-level servers, to which the kernel can delegate responsibility. System services, such as filesystem management, can be delegated to external servers. A new server can be added that has the same interface as an existing one, but with an entirely different implementation. A database management system can provide a new filesystem server that can be manipulated using existing system tools, but is implemented in a manner more appropriate for the DBMS' use.

  The Camelot distributed transaction processing system [EPPI91] takes advantage of this architecture, and provides a set of Mach processes to support nested transaction management, locking, recoverable storage allocation, and system configuration. Most of the mechanisms required to support transaction semantics are implemented at user level. The resulting system can be used by any application, not just clients of a Camelot data manager.

- *Mach VM:* The Mach virtual memory management system can delegate both storage and replacement policy decisions to user-level servers. In the Mach virtual memory management system [RASH88], *memory objects* are associated with a *pager*. The pager

is responsible for implementing the backing store for virtual memory.

The initial pager design did not allow pagers to control eviction policy. McNamee and Armstrong extended the interface to allow user-level control of page replacement policies [MCNAM90]. These extensions allow an application to specify the algorithm to use when selecting a page for eviction.

- *Vnode file system:* The VFS model [KLEI86] abstracts the implementation details of a file systems, allowing the rest of a Unix[1] kernel to interact with it through a fixed interface of approximately forty operations. VFS simplifies the task of adding a new filesystem type to a Unix system. The new filesystem is fully described by the implementation of its interface functions. These functions are then compiled into the kernel and added to a dispatch table.

2.2.2 Unified Resource Interface

- *Plan 9*, an operating system developed at Bell Laboratories ([PRES90]) offers a unified resource interface through the filesystem namespace, and the ability to extend the system by attaching a server process to the namespace.

2.2.3 Application Control of Policy

Two policy decisions are critical to the performance of an I/O intensive application: what data is cached in internal system buffers. and when it is read and written,

- Cao et al have investigated separating resource *allocation* decisions from resource *replacement* decisions in the context of a filesystem buffer cache [CAO94]. Allocation decisions (how many buffers are available for use by each process) are made by the kernel, based on its desire to share limited resources fairly. Replacement decisions (which buffer to evict) are delegated to the application program that "owns" the buffers.

- Transparent Informed Prefetching [PATT93] is based on the idea that application-level access pattern hints to the operating system can be more useful, and take less effort, than having the operating system attempt to infer future access patterns from past behavior. With better hints, the operating system can have a larger pool of outstanding requests over which to optimize I/O access, and can better overlap disk access with processing time.

- *Asynchronous I/O:* A non-blocking read can be used to prime the buffer cache with data before it is needed; a non-blocking write allows an application to queue a write request and continue to make progress while the write takes place [SOLA94].

  There are disadvantages to both techniques. When using non-blocking reads for read-ahead, the act of bringing in a block that will be needed in the future may push out a block needed in the present. A non-blocking write may fail (due to lack of disk space, or system crash), or a sequence of writes may not take place in the order requested by the application, which, in the event of a crash, can make recovery impossible.

2.2.4 Other Extensible Operating Systems

There are several other extensible operating system projects underway, including the SPIN project at the University of Washington [BERS94] and the Aegis project at MIT [ENGL94]. These operating systems allow applications to control general policy decisions.

Both SPIN and Aegis provide a minimal kernel, and focus on providing facilities for extending the kernel through the use of a system-specific extension language. Both are investigating runtime code generation to optimize kernel extensions.

The SPIN project concentrates on the idea of directly extending a Mach-style microkernel model by allowing servers to be installed in the operating system kernel. The developers of Aegis have stripped the kernel to a bare minimum, with only five system calls. Standard resources are built from these atomic components.

3 VINO Motivation

VINO is a new kernel architecture designed around the notion of a permeable barrier between the operating system and applications. Applications are empowered to direct resource management policy decisions.

VINO has three goals: to allow applications to specify the kernel policies that manage resources, to make kernel primitives accessible at user-level, and to provide a universal system interface designed around the acquisition of hardware and software resources. These goals combine to provide the flexibility and extensibility needed for effective support of a DBMS.

3.1 Application-Directed Kernel Policy

The problems cited in Stonebraker and discussed in the introduction result from applications having no input into the policies used to manage their resources. Suboptimal buffer management results when the kernel uses LRU globally; inappropriate process scheduling results when a single criterion is applied for all applications; poor I/O

[1] Unix is a trademark of X/Open.

performance results when the operating system assumes incorrect access patterns for data. The motivation for Application-Directed Kernel Policy is to enable applications to select their own resource management policies, leaving the kernel with the task of arbitrating between competing policies.

Chou and DeWitt [CHOU85] characterized disk access patterns for the different phases of operation of a relational database. Some blocks are reused, others are not. For example, when performing a nested loop join, the outer relation is scanned sequentially (so no caching is required), while the inner relation is scanned repeatedly, and so should be cached.

A standard assumption made by operating systems is that file access is either sequential or random. When a block is read from a file that is being sequentially accessed, an operating system may "read ahead" the following block or blocks. No prefetching is done on a file that is being accessed randomly.

The operating system makes the simplifying assumption that there are only two models of file access behavior. DBMS access patterns do not consistently match either one, and are not well served by the simplification.

Typically, a DBMS knows in advance which pages will be needed soon, and could inform the operating system of its future requirements. There have been numerous attempts to empower applications with the ability to control prefetching and page replacement [CAO94, MCNAM90, PATT93, HART92, APPEL91]. All of these approaches provide potential solutions, but the solutions apply only to buffer management and require the implementation of special purpose mechanisms. In VINO, application-specific tailoring is the norm, not the exception. *Every* resource policy decision can be modified at the discretion of the application.

3.2 Reusable Kernel Tools

The requirements of an operating system are in many ways the same as those of a database management system; both arbitrate access to shared resources. In the case of an operating system, those resources are things like physical memory, devices, and processor cycles. In a DBMS, those resources are data. Most conventional operating systems implement services such as synchronization primitives, log management, and buffer management. These services are buried inside the kernel, unavailable for application use. VINO is designed around the idea that any service used by the kernel should be accessible to user-level applications.

For example, a filesystem stores both data (file contents) and metadata (directory structure). When the metadata is modified, the modifications must be written out in such a way that the metadata can be recovered in case of a crash. Some filesystems carefully order writes (e.g. the Berkeley Fast Filesystem [MCKU84]); others use a write-ahead log for the metadata ([CHUT92, CHANG90]). Still other file systems use a log-structured representation for the entire file system [ROSE91]. While all these systems provide transaction semantics to metadata operations, none export this functionality to user-level.

In VINO, general-purpose log management code is used to facilitate the management of filesystem metadata, and is also available for use by applications. A DBMS does not have to implement its own log manager, and DBMS recovery is initiated as part of the standard system recovery process.

3.3 Universal Resource Access Interface

In order to make VINO's reusable kernel tools as widely applicable as possible, VINO provides a universal resource access interface to all system resources. The interface provides a simple, regular, abstract interface to system resources, independent of the representation of the resource.

Relational systems use this model to represent data as tuples, whether the data is stored in a B-tree, hash table, or other structure. Operating systems use this model as well; for example, the Network File System, the Berkeley Fast Filesystem, and the Berkeley Log-structured Filesystem [SELT93] provide the same interface to application programs, but the underlying representations are very different. Network sockets and disk files are both accessed through the use of file descriptors, although they are supported by entirely different implementations.

Some systems (e.g. Plan 9 [PRES90]) choose a single, compromise interface for all resources. The disadvantage of choosing one abstraction for all resources is that each must implement the complete interface, and none can extend it. VINO provides a more general, *hierarchical* resource model that allows behavior to be shared or factored out as appropriate, through the use of resource subtyping.

4 VINO Architecture

The VINO operating system architecture is composed of *resources* (objects, instances) and their *types* (classes). The resource type hierarchy is a static, single inheritance hierarchy, specified at system build time. To allow flexibility, individual resources can override or augment the implementation of individual resources at run-time.

Each resource type defines an interface consisting of a set of *operations* (functions member, message handlers) and *properties* (data members, slots). The resource type provides a default *implementation* for each operation, a piece of code that is run when the operation is invoked on a resource of that type.

The resource type hierarchy is completely specified at the time VINO is compiled; all VINO resources are de-

scribed by the standard types. The VINO distribution includes resource types for system resources (e.g. files, directories, memory, processes, threads, transactions, virtual memory page, physical memory page, synchronization primitives, and queue management).

4.1 Addition of New Types

We restrict the flexibility of the resource schema by requiring that it be specified at system build time. This provides performance and robustness superior to a system with a dynamic schema, because it allows static type-checking and operation dispatch. It is, however, a functional limitation, and we must weigh its benefits against the liabilities introduced.

A new resource type is introduced by subtyping an existing type. For example, the VideoFile resource type may be subtyped by QuickTimeFile. The new type could have the same interface as a VideoFile, or it may augment the interface, by adding new operations and properties.

A resource type is added to VINO by compiling any new or modified kernel code, and relinking the kernel. If the new type does not extend the interface of its supertype, it can be linked into the kernel without changing or recompiling any VINO source code. In this case, we can dynamically link the new resource type directly into the running kernel (just as device drivers are dynamically linked into current systems).

In a research environment, the requirement of relinking the VINO kernel to add a new resource type is, at worst, an inconvenience. In a production environment, we feel that new resource types will be added infrequently.

For example, the standard interface would support a device with a straightforward serial interface, such as audio or video. Support of *synchronized* audio and video streams would entail the addition of substantial new functionality, a new interface, and rebuilding the VINO kernel.

4.2 Overriding Implementation

Application control of policy is accomplished by overriding the default implementation of a resource. This is done by *grafting* a new implementation of an operation into VINO. The graft applies to a single resource, owned by the grafting application.

The graft is loaded dynamically into the kernel and will be used in the place of the default implementation provided by the resource type. Alternatively, a graft can be wrapped around the default implementation, adding a bit of pre- or post-processing to the standard behavior.

VINO places extension code into the kernel, rather than leaving it in user space (as is done in Mach). Research has shown that the cost of crossing protection boundaries (between user and system, or user-system-user, as in the case of a Mach external server) is very high [MOGUL91, CHEN93, GUIL91]. By placing extension code in the kernel, we decrease the number of protection boundary crossings, which improves performance. Additionally, graft implementations frequently use information gathered from the kernel; by placing grafts in the kernel, the bandwidth of the communication channel between the graft and the kernel is increased.

4.2.1 Graft Safety

When adding user-written code to a running system, the question of the *safety* of the code comes into question (not to mention the sanity of the system architects). How reasonable is it to allow random bits of code to be added to a running operating system?

The safety of a graft can be divided into two parts: its behavior relative to the application, and its behavior relative to the rest of the system.

In the first case, keep in mind that an application can only graft its own resources; a graft can not directly affect other applications. If a graft does not implement the standard semantics for an operation, only the application that installed the graft will suffer.

In the second case, we are concerned with whether the graft is benign or malicious with respect to the system. Code can attack the system in different ways; we are concerned with code that attempts to deny resources to other applications (e.g. through not giving up the processor in a timely fashion), that overwrites the kernel's private data structures, calls a privileged kernel function, or in some other violates system security.

We ensure safety of grafts through controlling the graft compilation environment. Grafts are written in C or C++, and compiled by our graft compiler, which is based on work underway at Harvard. The graft compiler inserts range checks on all memory accesses (including function calls). The checks ensure that a graft does not stray outside its bounds, or modify itself to do so.

A second scheme that ensures safety, *sandboxing*, was proposed by Wahbe et al. [WAHBE93]. Sandboxing assigns a range of memory to the code and data of each graft. Instructions are inserted into the code which force all memory references to fall within the assigned segments. Sandboxing masks faults instead of detecting them, but it is more efficient than range checking.

Code generated by the graft compiler is marked with an encrypted *fingerprint* that is effectively impossible to forge [RABIN81]. A fingerprint is a type of digital signature; if the fingerprint for a file is valid, its contents have not been modified.

In order to ensure that a graft does not keep a critical system lock for an extended period of time, each graft invocation is performed in the context of a transaction. When a lock is acquired for a critical system resource, the lock is assigned an expiration time. If the graft does not

release the lock before it expires, the graft's transaction is aborted.

The idea of a graft is not new; it is found in both operating systems (e.g. dynamic loading of device drivers in Solaris) and database systems (e.g. Illustra/POSTGRES and Thor). Our use differs in that grafts are used to not only augment the system, but to adapt VINO to meet the specific needs of an application. Grafts can be used to make small, *incremental* changes to VINO.

For example, a buffer management graft can be used to control the caching behavior for an application by modifying the buffer cache resource assigned to the application. The rest of the buffer management code, and the filesystem code, can be left unchanged. Under Mach, an application would need to develop a complete new server.

4.3 Dynamic Augmentation

In some rare situations, an application may need to create a resource that has operations not defined on its resource type. For example, an application may encrypt its data files; the graft code that reads the file will need to know the key. Because there is no interface on the File resource type to specify an encryption key, such a resource would need an augmented interface, with an additional operation.

We provide an escape mechanism that allows new operations to be added to a resource on-the-fly. These operations will not be invoked by the kernel, and applications have complete control over how they are used. A set of operations are provided by the root resource type that can be used to add, remove, and invoke dynamic augmentation operations.

4.4 Resource Allocation

One of the primary jobs of an operating system is to arbitrate and abstract resource access. Some devices, such as physical memory, are shared and preemptible; others, such as disk space or a serial port, are not.

Some applications require service guarantees, e.g. an application displaying real-time video using a double-buffered display needs to be scheduled thirty-two times a second and have physical memory large enough to hold two copies of the displayed image. A query processor can tune its join algorithm to the amount of physical memory available for its use, if it can assume that the memory, once allocated, will not be taken away.

Such applications can make *hard resource requests*, where no less than the minimum requested resources will be allocated, and once allocated, they will not be preempted. If a new *hard request* will exceed the physical resources of the system, VINO will not grant the request. A hard request can be thought of as application-specified entrance criteria; if the resource can not be allocated, the application can choose to not proceed.

In order to ensure fairness of allocation, an application must be privileged in order to make hard requests.

Applications with less stringent requirements can make *soft requests*, specifying a preferred minimum resource allocation. If the sum of the soft requests exceeds the system resources, VINO will arbitrate between the requesters, sharing the resources available.

5 VINO Services

VINO consists of a framework for implementing services, and a collection of services that implement standard operating system behavior. Many of these implement behavior that is shared by operating systems and databases. They can be used as-is, or augmented, to implement the underlying system support for a database management system.

5.1 Synchronization Primitives

The operating system's synchronization primitives are typically much simpler and more efficient than those provided to user-level applications. For example, System V-style user-level semaphores incur a large number of system calls and context switches while simple spin-locks are virtually free [SELT92]. VINO provides a kernel lock manager, accessible to resources.

In its simplest form, the lock manager provides spin-lock synchronization on memory locations, requiring kernel intervention only in the case of a contested lock. This interface is available both to the kernel and to applications.

As the resources being locked become more complex, so does the locking paradigm. The VINO lock manager supports *general-purpose hierarchical locking* [GRAY76]. For example, the file system typically requires locking on block, file, directory and file system levels. In most conventional kernels, this hierarchy is enforced by convention. In VINO, it is enforced by design.

We call the various levels at which locking may be needed the *containment hierarchy*. When a lock is requested from the lock manager, the manager consults the resource's containment hierarchy to determine if the lock may be granted. Applications using the lock manager can define alternate containment hierarchies, and make them available to the lock manager (e.g. a DBMS might create a logical containment hierarchy of database, relation, tuple, and field). Additionally, applications can create new instances of a lock manager that enforces alternate locking protocols (e.g. alternate deadlock handling, blocking vs. non-blocking, or new locking modes).

Finally, by integrating the kernel and user level locking systems, concurrency can be increased. For example, a DBMS running on a conventional Unix file system may

implement its own lock manager and issue multiple I/O requests to the same file. Unfortunately, the Unix file system exclusively locks the file during each I/O operation so that no concurrency is achieved even though the DBMS is already ensuring the integrity of the operation. In VINO, since the same lock manager handles both DBMS requests and I/O requests, locks held by the DBMS are strong enough to perform I/O and no additional locking is required by the file system.

5.2 Log Management

VINO provides a simple log management facility that is used by the file system and the transaction system, and accessible to applications as well.

A log resides on one or more physical devices. It can be created on a single device, or extended onto a second device (not necessarily of the same type as the first). For example, a DBMS might request a log that spans both magnetic disk and archive media (e.g. tape or optical disk). The kernel requests a volatile, in-memory log to support transactions on ephemeral data, such as process structures and buffer cache metadata. The file system requests a log that spans non-volatile RAM (NVRAM) and disk; file system log records are written first to NVRAM and are later written to disk in large, efficient transfers.

The key interface to the log facility is the read/write interface which provides the essential information for write-ahead logging (read by log-sequence-number and returning of log-sequence numbers by writes). It also supports a *WAL* operation for write-ahead log synchronization and a *checkpoint* operation for log reclamation and archival. As logs can expand and contract, the partitioning between log resources and other data resources is not static, but can change as system demands fluctuate.

5.3 Transaction Manager

The VINO kernel uses transactions to maintain consistency during updates to multiple related resources (e.g. a directory and its contents). For example, the carefully ordered writes of FFS can be reimplemented as a simpler series of unordered writes, encapsulated in a transaction.

The transaction interface supports the standard *transaction-begin*, *transaction-commit*, and *transaction-abort* interfaces. It accepts references to appropriate log and lock manager instances to use for each transaction. At transaction begin, a new *transaction resource* is created. This resource references the appropriate log and lock managers and is referenced by each protected update. Most kernel transactions are protected using a simple shadow-resource scheme with a log residing in main-memory (either volatile or non-volatile depending on the resources being protected). The mixing and matching of logging and locking components enables VINO to support arbitrarily complex transaction protocols.

Because the implementation of the transaction manager can be incrementally modified, different transaction sematics (e.g. as outlined in [BILI94]) can be implemented as needed by applications.

5.4 Name Service

Persistent objects are denoted by a *(manager, storage-id)* pair. A *manager* is a kernel resource that implements a file-like interface for each of the resources under its control. A manager can be thought of as a file system, and a *storage-id* as a file-id, relative to that filesystem. Equivalently, a manager can be thought of as a database, and the storage-id is the key of the corresponding record or object.

The *name service* maps a name in a global, hierarchical namespace to a *(manager, storage-id)* pair. Any resource that can be accessed via standard file-like operations (e.g. read and write) can be registered with the name service and made available through it.

The namespace is separate from the storage systems (managers); adjacent names in a directory can belong to different managers. For example, in the directory `/home/chris` one might find entries `time`, `sigmod.tex`, and `today-to-do`. The first would be read-only, managed by the `time-manager`; it responds to `read` requests by returning the current time of day. The second would be stored by a file manager, on a disk. The third would be a reference to a database, and when read, would present the result of a query that determined what was on today's to-do list.

5.5 File Service

The VINO file service is a manager that implements a fairly conventional FFS-style filesystem. It uses the VINO log manager to store its metadata, simplifying recovery in the event of a crash. Its persistent storage is provided by the *extent manager*, which arbitrates requests made for disk space from the different services that offer local persistent storage.

5.6 Memory Management

The VINO memory management system is based on the ideas of the Mach VM architecture (described in section 2.2.1), although its implementation differs considerably.

A *MemoryResource* is a collection of pages. As in Mach, it is backed by a file mapped into memory. It includes operations to read pages from and write pages to a backing store.

When a page fault takes place, VINO determines which MemoryResource (if any) is assigned to the virtual mem-

ory page containing the faulting address. A request is made of the MemoryResource, with the address at which to write the requested page.

Unlike a Mach pager, the MemoryResource is entirely in the kernel; when a page fault occurs (which causes a trap into the operating system), it is not necessary to go back across the protection boundary from the kernel to the user level.

Also, the Mach architecture requires that a new external pager be written for each kind of behavior needed. VINO allows each MemoryResource to use as much or as little of the standard implementation as is appropriate; the application need only override or augment the operations it wants to change.

The *AddressMapResource* is patterned after the Mach object of the same. Each address space has an associated AddressMapResource, which contains a mapping between physical pages and virtual memory pages. When VINO determines that a mapping needs to be removed (either because of a virtual memory fault, or because the number of physical pages assigned to the address space is being decreased), it invokes the ChooseVictim operation defined on AddressMapResource. By default, ChooseVictim selects the least recently used page, although an application can replace the implementation of ChooseVictim with the algorithm of its choice.

Note that, as in Cao's work (described in section 2.2.3), VINO retains control over the *number* of mappings allocated to an address space, but not the mappings themselves. The former behavior is not under the control of an application (in order to ensure fairness of allocation); the management of the mappings is delegated to each application.

6 VINO As a Platform For Database Development

Our motivation for developing VINO is to support resource-intensive applications, such as DBMS, better than conventional operating systems. To that end, we include high-level sketches of how the implementation of relational, object-relational, and object-oriented DBMS would change when built on VINO.

6.1 Relational Database

A relational database storage system consists of several components. We sketch how each would be implemented on VINO.

- *File Management:* data can be stored using the VINO file system, or a new storage manager (see section 5.4) can be developed that allocates and manages disk space. If the latter course is taken (because the file system allocation or management strategy is inappropriate for the DBMS), the FileSystemManager resource type can be subtyped to change it appropriately, or a new direct subtype of Manager can be written. Because the database storage is implemented using the Manager resource type, the database can be manipulated using standard utilities.

- *Buffer Management:* each proceess has, associated with it, one or more MemoryResources. A MemoryResource can be assigned to map a portion of the DBMS File resource into the address space of the DBMS. If the DBMS is larger than the virtual address space of the process, the MemoryResource type can be subtyped, or its implementation overridden, to "bank switch" pages of the underlying database into the process address space. The DBMS can control the decision of which disk pages to keep in physical memory by implementing a DBMS-specific ChooseVictim operation for the MemoryResource.

- *Log Management:* the VINO Log Manager can be used to maintain the database log. The physical storage of the log can be managed by the VINO file system, the DBMS, or a third service.

- *Lock Management:* the general-purpose hierarchical locking primitives provided by the VINO Lock Manager can be used directly by the DBMS.

- *Transaction Management:* transaction semantics can be supplied by the provided transaction manager, or extended to support extended transaction models.

- *Recovery Subsystem:* if the VINO transaction and log managers are used to implement the DBMS, no separate DBMS recovery subsystem is needed. After a crash, the VINO recovery system will traverse the database log and clean up after any transactions that were in progress at the time of the crash.

6.2 Object-Relational Database

The basic storage and consistency requirements of an object-relational DBMS (e.g. Illustra/POSTGRES) are similar to those of a relational DBMS. The primary difference between the two is support for data types not normally supported by relational systems.

Illustra/POSTGRES can be extended to support new data types through the use of "DataBlades."[2] A DataBlade is loaded into the client or server and implements the operations necessary to manipulate the data type.

[2]This is a reference to the razor-and-blades strategy of shaving equipment manufacturers; much more money is made from selling replacement blades than from the initial razor sale.

This model maps very cleanly onto the VINO Resource Type model. A relation would be modeled as a collection of Tuple resources; each Tuple would be a collection of Field resources. If a client application wished to extend the system, it would install a new Field resource type, either in the client or through negotiation with the server. The new type would implement the abstract behavior shared by all fields (i.e. get/set value, compare value with another Field), and would respond to requests to build and maintain type-specific indexes as needed.

6.3 Object Page Server

An object page server [DEWI90] serves pages of objects to client applications. Several current database systems (e.g. ObjectStore [LAMB91] and QuickStore [WHITE94]) are based on this model.

Object references are represented by virtual memory pointers. If an object is not in the client cache, its address is unmapped (invalid). When a client dereferences an unmapped pointer, a virtual memory fault takes place. The fault causes a trap to the kernel, which passes it on to an application-level signal handler. The signal handler determines that the fault is to an object not in the client's cache, and makes a request of a separate server process. If the page is not in the server's cache, the server reads the page from disk, and then passes it to the client handler. The handler installs the page in the client's memory, and restarts the client at the point of the fault.

The type of access made to the object is implicit, not explicit; the machine instruction that caused the fault was either a read or write instruction. In order to determine whether a read or write lock is needed, the handler must find the faulting instruction and inspect it. If a read lock is obtained, the page must be marked read-only; if the application then writes to the object, a second fault takes place, the client database code knows that the page was modified, and the read lock can be upgraded to a write lock.

Each fault, which entails multiple user to kernel domain crossings, is expensive. Under VINO, many of these domain crossings are unnecessary. The MemoryResource for the OODBMS virtual memory can be grafted to handle a fault directly; no dispatch to user level is needed. If the fault requires that the server be contacted, the graft can do this directly from the kernel. Modification of virtual memory protection is done without performing a system call.

It can take less time to copy a page of data over the network than to read it from a local disk [NELS88]. If the server has a page in its in-memory cache, a client need not write an unmodified page to disk when it is evicted. Each client cache would store only modified pages, and the server would be responsible for storing a single shared copy of the unmodified pages.

7 Conclusions

The operating system and database communities have been separately addressing the problems of inflexibility, uncooperativity, and irregular treatment of resources. Rather than ignore each other, the best way to accomplish the shared goals is to work together. A simple, regular architectural model is needed to support resource intensive applications.

When analyzing a hardware design, an engineer describes a badly aligned interface as an "impedance mismatch". We have the same problem in software: the operating system interface is inappropriate for DBMS, and for any other system that isn't the least common denominator.

We believe that the VINO architecture is a good step towards this needed cooperation, and is the right architecture for further research into DBMS and operating system structure.

8 References

References

[ACET86] Acetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the Summer Usenix Conference (July 1986).

[APPEL91] Appel, A., Li, K., "Virtual Memory Primitives for User Programs," *Proceedings of ASPLOS IV* (1991).

[BERS94] Bershad, B., C., Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, E. Sirer, "SPIN - An Extensible Microkernel for Application-specific Operating System Services", University of Washington Technical Report 94-03-03 (February 1994).

[BILI94] Biliris, S., Dar, S., Gehani, N., Jagadish, H. V., and Ramamritham, K., "ASSET: A System for Supporting Extended Transactions", *Proceedings of SIGMOD 94*, Minneapolis, MN (May 1994).

[CAO94] Cao, P., Felten, E., and Li, K., "Application-Controlled File Caching Policies", *Proceedings of the 1994 Winter Usenix Conference*, pp. 171-182 (June 1994).

[CHANG90] Chang, A., Mergen, M., Rader, R., Roberts, J., Porter, S., "Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors," *IBM Journal of Research and Development* v. 34, n. 1 (January 1990).

[CHEN93] Chen, B., and Bershad, B., "The Impact of Operating System Structure on Memory System Performance", *Proceedings of the 14th SOSP*, Asheville, NC (December 1993).

[CHOU85] Chou, H. and D. Dewitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proceedings of VLDB 85*, pp. 127-141, Stockholm, Sweeden (1985).

[CHUT92] Chutani, S., O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham, "The Episode File System," *Proceedings of the 1992 Winter Usenix Conference*, pp. 43-60 (January 1992).

[DEWI90] Dewitt, D., D. Maier, P. Futtersack, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for ODBMS", *Proceedings of the 16th VLDB* (1990).

[DION78] Dionysios, C. T., and A. Klug (eds.) "The ANSI/X3/SPARC CMBS Framework: Report on the Study Group on Data Base Management Systems", *Information Systems*, v. 3 (1978).

[ENGL94] Engler, D., M. F. Kaashoek, and J. O'Toole, "The Operating System Kernel as a Secure Programmable Machine" *Proceedings of the Sixth SIGOPS European Workshop* (September 1994).

[EPPI91] Eppinger, J, L. Mummert, and A. Spector, eds, "Camelot And Avalon, A Distributed Transaction Facility", Morgan Kaufman, San Mateo, CA (1991).

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., "Granularity of locks and degrees of consistency in a large shared data base," *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, pp. 365-394 (1976).

[GUIL91] , Guillemont, M., Lipkis, J., Orr, D., Rozier, M., "A Second-Generation Micro-Kernel Based UNIX: Lessons in Performance and Compatibility," *Proceedings of the 1991 Winter Usenix Technical Conference* (January 1991).

[HASK88] Haskin, R., Y. Malachi, W. Sawdon, and G. Chan, "Recovery Management in Quicksilver", *ACM Transactions on Computer Systems*, v. 6, n. 1, pp. 82-108 (February 1988).

[HART92] Harty, K., Cheriton, D., "Application-Controlled Physical Memory using External Page-Cache Management," *Proceedings of ASPLOS V*, Boston MA, pp. 187-192 (October 1992).

[ILLU94] *Illustra User's Guide, Illustra Server Releae 2.1*, Illustra Information Technologies, Inc., Oakland CA, (June 1994).

[KLEI86] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the 1986 Summer Usenix Conference*, pp. 238-247 (June 1986).

[LAMB91] Lamb, C., G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Communications of the ACM*, v. 34, n. 10, pp. 50-63 (October 1991).

[LISK93] Liskov, B., M. Day, and L. Shrira, "Distributed Object Management in Thor", in *Distributed Object Management*, Morgan Kaufmann, San Mateo, California (1993).

[MCKU84] McKusick, M., Joy, W., Leffler, S., Fabry, R., "A Fast File System for UNIX," *Transactions on Computer Systems*, v. 2 n. 3, pp. 181-197 (August 1984).

[MCNAM90] McNamee, D., and K. Armstrong, "Extending The Mach External Pager Interface To Accommodate User-Level Page Replacement Policies," *Proceedings of the 1990 Usenix Mach Workshop*, Burlington, VT (1990).

[MOGUL91] Mogul, J, and Borg, A., "The Affect of Context Switches on Cache Performance", *Proceedings of ASPLOS IV* (1991).

[MYERS93] Myers, A., "Resolving the Integrity / Performance Conflict" *Fourth Workshop on Workstation Operating Systems*, pp. 156-159 (October 1993).

[NELS88] Nelson, M., Welch, B., and Ousterhout, J., "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, v. 6, n. 1 (February 1988).

[ORA89] *Oracle Database Administrator's Guide*, Oracle Corporation, 3601-V6.0 (April 1989).

[PATT93] Patterson, R., G. Gibson, M. Satyanarayanan, "A Status Report on Transparent Informed Prefetching", Operating Systems Review, v. 27, n. 2, pp. 21-34 (April 1993).

[PRES90] Presotto, D., R. Pike, H. Trickey, and K. Thompson, "Plan 9, a Distributed System, *Proceedings of the Spring 1991 EurOpen Conference* (May 1991).

[RABIN81] Rabin, M., "Fingerprinting by Random Polynomials", Harvard University Center for Research in Computing Technology TR-15-81 (1981).

[RASH88] Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, "Machine-Independent Virtual Memory Management for Page Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*, v. 37, n. 8 (August 1988).

[ROSE91] Rosenblum, M., Ousterhout, J. K., "The Design and Implementation of a Log-Structured File System," *Transactions on Computer Systems* v. 10, n. 1, pp. 26-52 (February 1992).

[SELT92] Seltzer, M., Olson, M., "LIBTP: Portable, Modular Transactions for UNIX," *Proceedings 1992 Winter Usenix Conference*, San Francisco, CA, pp. 9-26 (January 1992).

[SELT93] Seltzer, M., K. Bostic, M. K. McKusic, and C. Staelin, "An Implementation of a Log-structured Filesystem for UNIX", *Proceedings of the Winter Usenix Conference* (January 1993).

[SKAR94] Skarra, A., "Using OS Locking Services to Implement a DBMS: An Experience Report", Proceedings of the Summer USENIX Conference, Boston MA, pp. 73-86 (June 1994).

[SOLA94] *Solaris 2.3 User's Guide*, Sun Microsystems (1994).

[STON81] Stonebraker, M., "Operaing System Support for Database Management", Communications of the ACM, v. 24, n. 7, pp. 412-418 (July 1981).

[STON87] Stonebraker, M., and L. Rowe, "The POSTGRES Papers", Memorandum No. UCB/ERL M86/85, Electronics Research Laboratory, University of California, Berkeley (June 1987).

[SULL91] Sullivan, M., and R. Chillarege, "Software Defects and Their Impact on System Availability – A Study of Field Failures in Operating Systems", *Digest 21st International Symposium on Fault Tolerant Computing* (June 1991).

[SYB90] *Sybase Administration Guide*, Sybase Corporation, 3250-4.2 Rev. 3 (May 1990).

[WAHBE93] Wahbe, R., Lucco, S., Anderson, T., and Graham, S., "Efficient Software-Based Fault Isolation", *Proceedings of the 14th SOSP*, Asheville, NC (December 1993).

[WHITE94] White, S., DeWitt, D., "QuickStore: A High Performance Mapped Object Store", *Proceedings of SIGMOD 94*, Minneapolis, MN (May 1994).