A WAY FOR SEAMLESS INTEGRATION OF DATABASES AND OPERATING SYSTEMS

Thomas Schöbel-Theuer, University of Stuttgart
Universitätsstr. 38, D-70569 Stuttgart

schoebel@informatik.uni-stuttgart.de

ABSTRACT: *We present a novel common architecture for both operating systems and databases. Seamless integration will blur differences from user's viewpoints and allow for easier merging of filesystem data and database data, as well as for uniform treatment of either sort of data with both OS and DB access methods. This will simplify management and retrieval of data, reduce training costs, open data treasures for better exploitation, and simplify integration of applications which rely on different system interfaces. Seamless integration can achieve both* representation transparency *and* interface diversity *in a uniform way, less complicated and more capable than glueing together conventional DBs and OSs.*

Keywords: *operating systems, databases, software architecture, statelessness, synergy effects, representation transparency, interface diversity.*

1. INTRODUCTION

Operating systems (OSs) and database systems (DBs) have developed rather independently in history. Traditionally, DB designers have either started to "write a new, simpler and 'vastly superior' operating system", or they have "extend[ed] the basic operating system to have the desired function" [11]. Most DBs have taken the first choice (basically using only device drivers from the OS and acting as normal user processes); only a few like IMS have taken the second choice, basically delivering a functionality which is not in direct competition with most OS services. DB designers have expressed their needs a few times and criticized inadequate OS support [21, 25, 22], which led to OS completions for *some* of their most critical desires, but mainstream OSs like UNIX are still separate from DB systems. Today, OSs and DBs are regarded as two different subfields of computer science with little interactions.

We propose a third way: seamless integration of OSs and DBs by use of a novel *common architecture* for both fields. This is fundamentally different from attempts to "glue together" both worlds.

The basic idea is to use a *universal* abstraction called "nest" on all levels of the system, which may not only emulate classical OS abstractions such as files, whole filesystem subtrees, process images and others, but may also emulate relational DB tables. Nest instances are transformed into other nest instances by functional units called "bricks". In contrast to object oriented (OO) style, bricks may be implemented *stateless*, leading to enormous simplification of dynamic reconfigurations such as data migration or process migration in a network of computers.

A key feature of our architecture is the ability to handle *multiple views* on the system. As an example, there may be location transparent views which abstract away from concrete instance locations in a computer network, views hiding technical details such as connector objects, views providing different programming interfaces, or virtual views creating a totally different higher-level picture of the system. Although we have originally developed views for our OS design, they naturally correspond to DB views.

Using a common architecture for both DBs and OSs allows for both representation transparency and interface diversity, in a less complex way as with known methods like wrapping [17] or glueing together conventional DB and OS implementations [3]. Our architecture allows *orthogonal combinations* with location transparency, hardware transparency, fault tolerance and other forms of (re)configuration automation, leading to increased functionality, simplicity, stability, security, and decreased overhead.

The paper is organized as follows: section 2 introduces basic concepts by examples from our operat-

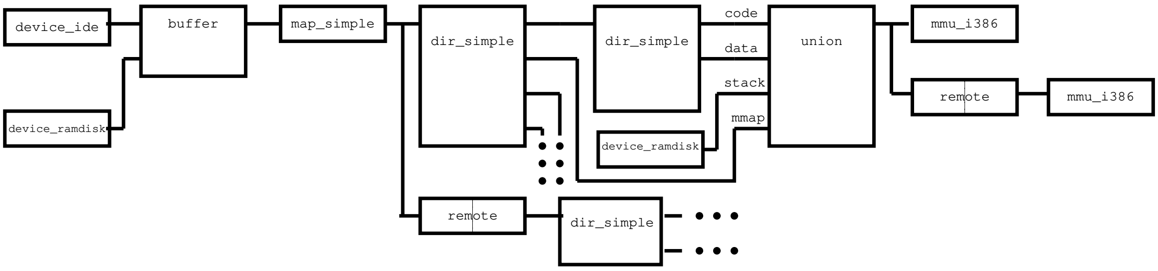


Figure 1: stripped-down OS scenario

ing system design. Section 3 sketches emulation of basic database functionality. Section 4 discusses the criticism from Stonebraker [21]. Section 5 shows how databases and operating systems could be seamlessly integrated, and section 6 concludes the paper.

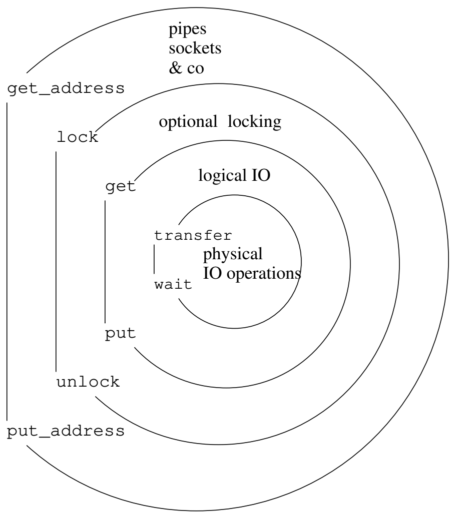
2. A DESIGN FOR OPERATING SYSTEMS

Our original motivation comes from OSs, for which we have developed a novel architecture [19, 18]. According to the example scenario in figure 1, it may be characterized as a "pipes and filters style" in the sense of software architecture [20].

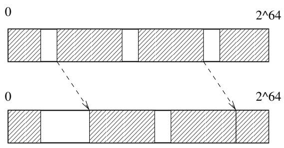
The wires in figure 1 may be viewed as "transportation channels" which *logically* transport instances of a universal address space abstraction called "nest". The boxes are called "bricks" and may be characterized as "transformers" between nest instances. They are depicted with inputs on their left and outputs on their right, similar to functional units in electrical engineering [14] or automation control [15]. Wires are directionally drawn from left to right. By dynamic wiring, we implement anonymous connection-oriented directional communication (as opposed to OO, which employs connectionless undirected communication, often on known partners). Communication normally goes from right to left, in the opposite direction of logical transportation.

Our "pipe and filter style" differs from known implementations of that style in the OS area, such as UIO [5] or stackable filesystems [13], in a number of ways. First, it has no "consuming" semantics like pipes, where processing of data would "destroy" the old data and produce a new version instead. Rather, a brick adds to the ways we may look at the system, by providing a new *view* on the data: both the old view and the new view provided by the transformation will exist in parallel; the "old" view may for example be used by parallel wiring to other "consumers" or "clients". Second, the nest is a *universal* (but nevertheless rather simple) abstraction for *all* layers of the whole OS, not limited to filesystems (in contrast, stackable filesystems are based on conventional filesystem abstractions such as `inodes`, `vfs`, directory (sub)trees, etc). We take advantage of the fact that bricks may be instantiated dynamically at runtime, thus resembling the dynamic nature of filesystem subtrees directly by recursive instantiation of `dir_*`-bricks.

Nest instances are used for modeling *logical address spaces* (similar to virtual address spaces, but independent from MMU hardware). A nest instance represents a *sparse* address space with arbitrary holes, managing data blocks in multiples of an arbitrary `transfer_size` value. Here is an overview on the elementary operations of the nest interface:

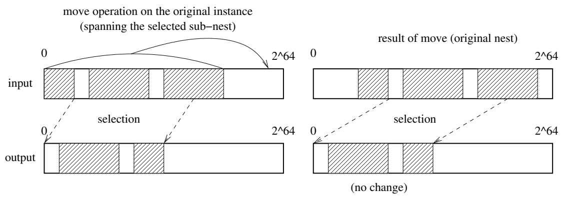


The elementary operations are organized as a layered system, where each layer consists of two operations which should be used pairwise. On the innermost layer, `transfer` and `wait` can be used to perform both synchronous and asynchronous IO on known physical addresses of raw data blocks; thus it is called *physical IO*. The next layer comprises operations `get` and `put` and is called *logical IO*, which translates from logical addresses to physical addresses similar to conventional buffer caches. On the next layer, read-locks and write-locks are implemented to solve the problem of *mutual exclusion* which occurs when multiple inputs are connected to a single output (parallel wiring). The outermost layer solves the problem of atomar reservation in the logical address space; it is used for parallel operations on pipes and sockets. Details may be found in [18].

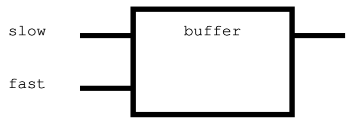


In extension to known address space abstractions, the nest interface has an additional operation `move`. It is used for *transparent moving* of data blocks in the logical address space. Moves are not implemented by heavy copying of data, but rather by *modifying* the internal association from logical to physical addresses. When implemented in a smart way, the performance of `move` operations is expected to be quite satisfying. Move operations are particularly useful for reorganizations in logical address spaces, e.g. when keeping whole filesystem images or filesystem subtrees within a nest instance. Files and filesystem subtrees may be kept *contiguously* (or *near*ly contiguous) in the logical address space. When space requirements are changing, we may simply create new space or delete unused space by means of `move` operations.

As an example for *transparent* `move` operations, look at a `selector` brick, which selects a part of the `input` space and makes it available at the `output`, but with logical addresses restarting from logical address 0:



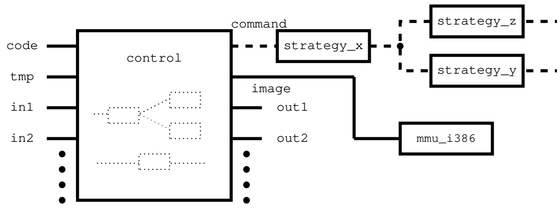
When a `move` operation spans at least the whole selected part, the output is not shifted in the same way as the input, but rather the "anchor" of the selected part is moved according to the shift distance, such that the logical "contents" of the output remains unchanged (*invariant* or *transparent* move). Transparent moves are particularly useful for emulating filesystem semantics in the presence of open files. Some details of the internal workings of `dir_simple`-bricks may be found in [18].
A very short overview on some basic brick types for OSs as proposed in [18]: `device_*` resembles device drivers, `buffer` corresponds to buffer caches, `map_*` implements sparsity and the `move` operation and solves fragmentation (locality of access), `dir_*` resembles flat filesystem directories, `union` "concatenates" or "mounts together" multiple nest instances to a single one, `mmu_*` acts as device driver for the MMU hardware to execute a "process image", `adaptor_*` translates between different `transfer_size` values or access models, `cow` implements the copy-on-write strategy, `remote` uses the client-server-paradigm to export a nest instance to a remote site in a computer network, and `mirror` implements distributed shared memory and/or replication of a nest instance. Further brick types like `pipe` or `socket` may be added. With `transaction`, we may introduce ACID semantics [12] for any part of an OS.
Bricks may be implemented *stateless*. A brick is called stateless, if it may be de-instantiated at any time provided that currently no activity is going on inside it, and later be re-instantiated (with the same wire connections), such that there is no observable difference in behaviour from the outside. As an example, look at a `buffer` brick, which solves the problem of access gaps in memory hierarchies:



Caches use internal data structures such as hash tables for keeping a transient mapping from logical addresses to physical addresses (of the cached data blocks). A stateless `buffer` implementation will keep its internal state in the `fast`-input, as well as the data blocks which are to be maintained by the cache. If *all* state information is always kept in the `fast`-input instead of inside the instance, the buffer may be de-instantiated at any time when there is no activity, and re-instantiated without even a noticeable effect on runtime caching behaviour. If statelessness is applied to any brick type, we get a network of instances which delegate the responsibility for state keeping transitively to their predecessors until some `device_*` is reached, which itself delegates it to the hardware (for performance reasons, so-called *pseudo-stateless* buffers may be inserted to keep some state for some limited time). Statelessness allows for enormous simplification of reconfiguration, such as process migration on a network of computers.

The property of statelessness forms a major difference to OO style of thinking, where both state and behaviour are regarded as essential for objects. Bricks are rather similar to *components*, which are also stateless according to Szyperskis definition in [24], but components cannot be instantiated like our bricks (we may even instantiate them *recursively*).

Now we look at the way how instances are generated.



The idea is to create and maintain brick instances of any type by a special brick called `control`. However, a `control` instance does not create other brick instances directly, it rather creates an `image` which may be thought of a "process image" which "contains" the instances and their wiring. In order to really "execute" the instances network, some `mmu` instance like `mmu_i386` has to follow after (possibly indirectly), similar to the method "normal" "processes" are brought to execution in figure 1. This leads to a separation between the controlling instance which controls the wiring (relations between instances) and the actual environment where the instances are executed; instances are disallowed from modifying their relations directly (for example, cyclic wiring may be disallowed [8]).

Moreover, this leads to *another* separation: the *commands* for instantiation and de-instantiation are issued on "control lines" which are depicted as dashed wires. Over these control wires, one can get information on the instantiated network (such as the wiring graph structure) as well. This means, a dashed control wire provides a *view* on the system structure and the interrelations between instances. There may exist multiple views *in parallel*. In the example, there exist different `strategy_*`-instances which may *transform* between views, or even create virtual views which do not exist in "reality". Moreover, the wiring of `strategy_*`-instances may itself be controlled by `strategy_strategy_*`-instances and so on, but at some arbitrary level these control levels should either terminate by a level which controls itself, or at a level where never any modifications are neccessary (note that such a level is originally instantiated by a bootstrap mechanism which is neccessarily outside the system; see [18]).

What are the benefits of separation of control over instances?

As an example, `strategy_transparency` may provide location transparency, by automatically inserting `remote`-instances into the system wherever several hosts must be spanned on a computer network, and by providing a single virtual system image. The automatically inserted `remote`-instances may be hidden for users of `strategy_transparency`, so that they get the virtual impression that no network would exist at all, and as if everything would execute on a single "virtual computer".

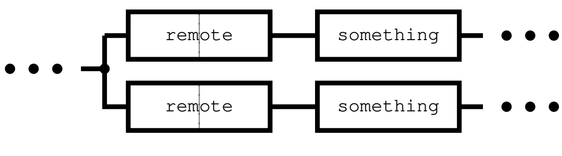
As another example, we could provide different OS *personalities*, such as different sets of system interfaces (by relaxing the use of single nest interface type at certain well-chosen points), to allow for heterogeneous mixing of applications which have been written for different OS platforms. These views may not only exist in parallel, but may be combined with each other, e.g. with network transparency.

As another example, we may employ code translation similar to *code morphing* [1] to create hardware platform transparency. On the instance level, executable code for the Intel 80x86 processor may be converted by `morph_i386_sparc` to code for the SPARC processor. Such code morphing bricks may be inserted automatically by a `strategy_morph`-level, whenever applications are executed on SPARC machines instead of Intel machines, or vice versa (e.g. when process migration crosses hardware architecture borders). In general, hardware platform independency requires not only hardware-specific instances such as `mmu_sparc` in place of `mmu_i386`, but may require further conversions of data contents such as byte order conversions (for suggestions on generic type systems describing data formats, see [18]). On the "virtual computer" level, we may provide a view in which hardware specific details are completely hidden. When combined with network transparency, it should not make any difference whether one buys a SPARC or an Intel computer and connects it to the network.

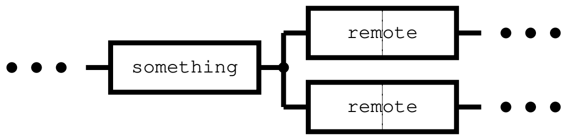
Note that that often different views are needed in parallel. For example, a service technician may need hardware-dependent views for locating a faulty device, and a network supervisor may be interested in views of the physical network configuration.

Further examples for `strategy_*` are automatic adaptation to varying workloads (load balancing), or to unpredictable faults (fault tolerance), or to specialized needs such as multimedia support or transaction support, or automatic insertion of `adaptor_*` where necessary, or centralized enforcement of security policies by inserting `check_*` instances, just to mention a few possibilities.

Finally, we look at another example where a key property of OO is relaxed: the notion of *identity* is softened. In a wired network based on stateless brick instances, state is not represented by brick instances, but rather kept in nest instances. Even that is not the full truth: there may exist nest instances which are *equivalent* to each other (i.e. they form *aliases* in logical sense), although they are not identical. As an example, look at the following picture:



Because of the properties of statelessness, this is *functionally equivalent* to the following network:

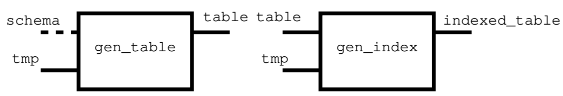


Which of these configuration variants will deliver better performance in practice? There is no general answer. It may depend on subtle properties of network bottlenecks, on required throughput before and after the `something`-instance(s) (which may be different), on the CPU load the work in `something` may produce in total, and on the synchronization overhead between two paralleled `something`-instances, among many other influences. In one case the first variant may be better, in another case the second. So what to do?

We may create `strategy_*`-bricks transforming one configuration variant into another, depending on knowledge on the problem domain, on environmental properties, and on actual measurement data (e.g. for current load). Such transformations may for example be governed by rule-based methods, on an automatic basis.

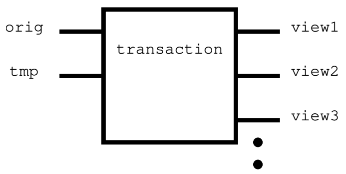
3. EXTENSION TO DATABASES

The basic idea for implementing DB functionality with wired brick networks is simply to keep sets of objects, e.g. relational DB tables, in nest instances. A table of fixed-length records may be directly emulated using a `transfer_size` equaling to the record length. Since variable-length records or record fields such as BLOBS (binary large objects [9]) naturally correspond to files in OSs, we may use the concepts from the previous section for them (but take care for excellent performance even for extremely small objects; for some suggestions see [18]).

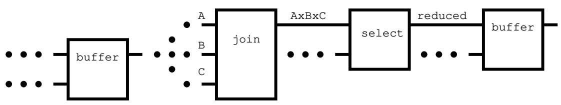


Tables may be generated from external `schema` specifications with `gen_table`, keeping necessary state in its `tmp` input. A relational table will normally occupy a contiguous area in the logical address space of a nest instance. Insertions and deletions may be directly carried out using `move` operations; the implementation should ensure data consistency by allowing only *correct* operations which are

compatible with the schema definition. Adding multiple indexes is possible with gen_index, which adds a (virtual) table of record numbers (at a separate contiguous area in logical address space), which is sorted by an external sorting criterion. The primary index can be omitted if we keep the table always sorted according to the primary index; the presence of the move operation allows for different solutions from conventional designs (note that we are just discussing the *presentation* at the interface level; the internal realisation may employ totally different methods such as B-trees or hashing).
Basic DB operations may be implemented as bricks emulating relational algebra operations, such as select, join, and projection. By dynamic wiring to complex brick networks, any expression from relational calculus may be emulated.



Transaction support may be added by transaction_* bricks. A transaction is nothing other than an *isolated view* on an original nest instance which is independent from other views and obeys the ACID properties [12, 7]. An alternative way of introducing transaction semantics into the nest interface may be found in [18]; a description would exceed the space limits for this paper.
Translation of SQL statements to brick networks is possible with strategy_sql or similar bricks on the strategy_* level. Query optimization may be done either directly in strategy_sql, or in subordinated bricks like strategy_optimize. The latter should be preferred when remote or mirror instances are to be inserted for emulation of distributed DBs; by dynamic choice of insertion points for remote, distributed query optimization and load balancing may be performed. Even dynamic insertion of additional buffer instances may be worthwile in some cases:



Here we look at a *pathological* example of a SQL statement which produces a join of three tables A, B and C, and applies a selection predicate which depends *non-uniformly* on the product records as a whole, not on their components (otherwise query optimization could first execute select before applying join). When the sum of tables A, B and C fits in the lower buffer instance, we get no heavy performance problem when join traverses them repeatedly. However, when select throws out most of the product records, only a small number of records will survive at the reduced output. When the result is read many times by many consumers, the huge join product would be repeated each time, leading to high CPU load. By appending a second buffer instance at the end of the chain, repeated evaluation of the product can be avoided in many cases, leading to a drastic reduction in overall CPU consumption. The example shows that recursive instantiation of bricks is a very powerful concept.

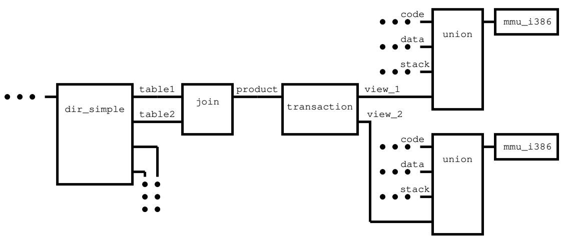
4. FUNCTIONALITY AND PERFORMANCE

Stonebraker has criticized contemporary UNIX support for DBs in 1981 [21] and argued for separation of DB implementations from OS implementations. In our opinion, all the points of his criticism can be resolved with an appropriate common architecture, if people from both fields are willing to work together. Our OS architecture [18] already includes facilities for DB support such as various lock types, speculative locking, IO priorities including background IO, partial ordering of IO requests, hints for various cache replacement strategies, generic data type descriptions, generic and extensible support for metadata, and generic support for different access models, which should be sufficient for most basic DB needs. However we are natively working only in the field of OS design, and we might have overlooked some sophisticated DB needs. Thus our architecture should be jointly revised and improved in close cooperation with experienced DB designers.
Many of Stonebraker's points refer to the inflexibility of conventional OS designs, in particular to drawbacks of monolithical OS kernels. Our architecture allows emulation of almost any static OS architecture by dynamic (re)configuration. For example, we may emulate the configuration and protection models of monolithical kernels, microkernels [16], exokernels [10], single-address-space models [4], nested

virtual machines [6], and hybrid configurations by different placement strategies for brick instances in different `control` instances and by composition of different `image` outputs of multiple `control` instances to form a single logical `image`.
Another flexibility of our architecture arises from separation between *interface representation* and *communication mechanisms*. In extension to ideas from [23], the wires in our model may be simultanously implemented as (1) RPC, (2) LRPC [2], (3) indirect procedure calls, (4) direct procedure calls, and (5) macro / inline expansion. Although (4) and (5) can be executed only on static brick subnetworks, we may employ *dynamic linking* or *dynamic code generation* (when sourcecode or some intermediate code is available) for replacement of subnetworks by dynamically generated bricks, controlled by some automatic `strategy_replace`. We hope that the latter will lead to considerable and competitive performance boosts when combined with intelligent address space placement strategies for brick instances.

5. SEAMLESS INTEGRATION

Seamless integration between OSs and DBs means more than just using common infrastructure like `buffer`, `remote`, `mirror` and `transaction`. As an example, we may directly map relational tables into "process images" at the instance level, similar to memory-mapped files. This goes beyond the capabilities of supplementary connections between filesystems and DBs such as DataLinks [3]. The following picture is somewhat simplified, for instance we left out `adaptor_*` for adjustment of different `transfer_size` values:



As another example, we may create pseudo-`dir_*` bricks translating DB tables into "filesystem directories" containing small virtual files representing records; the granularity could be even lowered by representing record fields as tiny files in another type of pseudo-directories. These files could then be browsed by user tools like file managers or accessed via an ftp server.
Conversion of collections of files to virtual DB tables is also possible, in order to apply relational algebra operations to them.
At the `strategy_*` level, we may combine both possibilities of treating filesystem data as DB tables, and DB tables as filesystem data for *representation transparency*. This means, we can provide a virtual view where it does not matter in which representation data was originally fed into the system. Moreover, processing of data is possible with both classical filesystem operations and with DB operations, e.g. SQL statements (interface diversity). As a consequence, DBs and OSs will become indistict at a higher level. Moreover, combination with network transparency, fault tolerance etc (see section 2) at a rather fine-grained level of control is *orthogonal*; the latter property would be extremely difficult to achieve with wrappers [17] or DataLinks [3]. Our approach of using *universal* and *generic* brick types for a wide spectrum of applications allows for reduction of redundancy in the overall system; nevertheless specialized needs can be served with different `x_*` brick types where necessary.

6. CONCLUSIONS

We have presented the idea and some design examples for a common universal architecture for both OSs and DBs. A common architecture allows for *seamless* integration of OSs and DBs, leading to new capabilities which would be hard to achieve by glueing together conventional systems, or would lead to unnecessary (performance) overhead. For example, representation transparency and interface diversity may be *orthogonally* combined with location transparency or hardware architecture transparency.
Our architecture provides for some extraordinary features: statelessness, a universal abstraction called "nest" for uniform use throughout OSs and DBs at all levels, dynamic composition of bricks controlled by a separate `strategy` level, and many detailed solutions differing from conventional architectures.
Implementation and performance studies are some of our next goals. Both DBs and OSs have undergone years of research and improvements; achieving a similar level with a new architecture will require substantial effort and cooperation from many places.

References

[1] *Code Morphing*. http://www.transmeta.com/technology/architecture/code_morphing.html.

[2] BERSHAD, BRIAN N. and OTHERS: *Lightweight Remote Procedure Call*. Transactions on Computer Systems, 8(1):37–5, 1990.

[3] BHATTACHARYA, SUPARNA, C. MOHAN, KAREN W. BRANNON, INDERPAL NARANG, HUI-I HSIAO and MAHADEVAN SUBRAMANIAN: *Coordinating backup/recovery and data consistency between database and file systems*. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 500–511. ACM Press, 2002.

[4] CHASE, JEFFREY S. and OTHERS: *Sharing and Protection in a Single-Address-Space Operating System*. Transactions on Computer Systems, 12(4):271–307, 1994.

[5] CHERITON, DAVID R.: *UIO: A Uniform I/O System Interface for Distributed Systems*. Transactions on Computer Systems, 5(1):12–46, 1987.

[6] CREASY, R. J.: *The Origin of hte VM/370 Time-Sharing System*. IBM Journal of Research and Development, 25(5):483–490, 1981.

[7] DATE, C. J.: *An Introduction to Database Systems*. Addison Wesley, 1995.

[8] DIJKSTRA, EDSGER W.: *The Structure of the "THE" Multiprogramming System*. CACM, 11(5):341–346, 1968.

[9] ELMASRI, RAMEZ and SHAMKANT B. NAVATHE: *Fundamentals of Database Systems*. Addison Wesley, 1995.

[10] ENGLER, DAWSON R., M. FRANS KAASHOEK and JAMES O'TOOLE: *Exokernel: An Operating System Architecture for Application-Level Resource Management*. Symposium on Operating System Principles, pages 251–266, 1995.

[11] GRAY, JIM: *Notes on Data Base Operating Systems*. In FLYNN, M. J. and OTHERS (editors): *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 394–481. Springer-Verlag, 1978.

[12] GRAY, JIM and ANDREAS REUTER: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] HEIDEMANN, JOHN S. and GERALD J. POPEK: *File-System Development with Stackable Layers*. Transactions on Computer Systems, 12(1):58–89, 1994.

[14] HOTZ, GÜNTER: *Schaltkreistheorie*. De Gruyter, 1974.

[15] *IEC Standard 61131-3*. http://www.holobloc.com/stds/iec/sc65bwg7tf3/html/news.htm.

[16] LIEDTKE, JOCHEN: *On μ-Kernel-Construction*. Symposium on Operating System Principles, pages 237–250, 1995.

[17] ROTH, M. and P. SCHWARZ: *Don't Scrap It, Wrap it! A Wrapper Architecture for Legacy Data Sources*. Proc. VLDB Conference, 1997.

[18] SCHÖBEL-THEUER, THOMAS: *Eine neue Architektur für Betriebssysteme*. Unveröffentlichtes Manuskript einer Monographie, 2003. Erhältlich auf Anfrage bei schoebel@informatik.uni-stuttgart.de.

[19] SCHÖBEL-THEUER, THOMAS: *Skizze einer auf nur zwei Abstraktionen beruhenden Betriebssystem-Architektur: Nester und Bausteine*. Arbeitspapier und Vortrag auf dem Herbsttreffen der GI, Fachgruppe Betriebssysteme, Berlin, 7. – 8.11.2002.

[20] SHAW, MARY and DAVID GARLAN: *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[21] STONEBRAKER, MICHAEL: *Operating System Support for Database Management*. CACM, 24(7):412–418, 1981.

[22] STONEBRAKER, MICHAEL: *Problems in Supporting Data Base Transactions in an Operating System Manager*. Operating Systems Review, 19(1):6–14, 1985.

[23] STROUSTRUP, BJARNE: *On Unifying Module Interfaces*. Operating Systems Review, 12(1):90–98, 1978.

[24] SZYPERSKI, CLEMENS: *Component Software*. Addison-Wesley, 1998.

[25] TRAIGER, IRVING L.: *Virtual Memory Management for Database Systems*. Operating Systems Review, 16(4):26–48, 1982.