Making Database Systems Fast Enough for CAD Applications

David Maier

Technical Report No. CS/E 87-016

December 15, 1987

Making Database Systems Fast Enough for CAD Applications

David Maier

Technical Report CS/E-87-016
15 December 1987

Computer Science and Engineering
Oregon Graduate Center
19600 S.W. von Neumann Drive
Beaverton, Oregon 97006-1999

Submitted to ***Object-Oriented Concepts, Applications and Databases,*** **W. Kim and F. Lochovsky, Eds.**

Making Database Systems Fast Enough for CAD Applications

David Maier

Oregon Graduate Center
Dept. of Computer Science & Engineering
19600 NW von Neumann Dr.
Beaverton, OR 97006

Why are database systems an infrequent component of CAD systems? Lack of modeling power and performance. Record-based models aren't up to handling complex design structures for VLSI or mechanical CAD. It's hard to design an ALU in your check register. The encoding involved greatly complicates application programs, and leaves too much room for misinterpretation of data. For flexibility, most CAD systems perform their own data management on top of the OS file system. Of course, a file system has no understanding of the structure of the data items in a design, and so is powerless to help with integrity or specialized storage mappings. Also, the data management features in a file system are minimal.

Those CAD systems that are built atop relational databases use them almost exclusively for selection, sometimes for projection, but perform or precompute joins in the program memory of the design tools. Such tools read the approriate chunk of the database at start up, and build internal record and pointer structures from it in virtual memory. Thus, a design session starts by copying one part of the disk into another. At the end of the session, the internal structures are converted back to tuples and written to the database. This start-up overhead gives a batch-processing flavor to doing design. Consider a VLSI designer who discovers a glitch in a layout while using a design rule checker. He or she must load the design into the layout editor, make the change, dump the design back into the database, then reload the design into the checker. The whole process can take up to one-half hour, simply to reroute a wire. Furthermore, the virtual memory knows little about the structure of the design objects, and can't be expected to give good paging performance.

Such CAD systems are using a DBMS as an index package to support associative access. These systems forgo the other data management features of the DBMS, such as recovery, associative access, integrity checking and buffer management, during a design session. The ubiquitous reason is "Performance! Commercial database systems aren't fast enough to support simulators and interactive design tools."

A crop of object-oriented database systems is emerging that greatly surpasses conventional record-based systems in modeling power [Atwood 1985, Banerjee, et al. 1987, Dittrich, et al. 1986, Katz and Chang 1987, Landis 1986, Maier, et al. 1986, Wegner and Zdonik 1986]. These systems overcome the data model objections to using DBMSs for design support—but will they prove fast enough? The following sections give my opinion on what conventional database systems are too slow at, why they are slow at it, and why OODBs could deliver the required performance. My remarks are directed at relational DBMSs, but most carry over to hierarchical and network models. I conclude with some areas for more research.

What's Too Slow?

Certainly conventional relational systems are no slouches at associative retrieval. Their indexing routines are among the most sophisticated and highly tuned of any class of software systems. Associative access to disk isn't the bottleneck—as I pointed out, when CAD tools do use a DBMS, it's for selection. No, conventional databases are too slow at fetching and storing individual fields. A typical design task is unlike most data processing transactions. The latter either involve getting a few tuples from a relation and updating them, or selecting large groups of tuples from one or more relations and performing similar operations on the lot of them: taking a join to generate a report, updating a field in each to post interest. The design task also starts with a selection—to pull out the pieces of a design of interest—but then continues with many dissimilar fetch and store operations: move this strut a little to the right, propagate this signal to all inputs connected to the output of that NAND-gate. The access paths on the selected data follow the connectivity of the real-world entities, not the logical structures of the database.

The ideal is a database system in which field access is as fast for database items as for value in program memory. Field access in program memory can take as few as one or two machine instructions, if the proper addressing modes exist. While that speed may be unobtainable for database items, I believe coming within a a factor of 10 is possible, rather than the factors of 1000 or 10,000 seen now.

Why It's Too Slow

CAD tools use record structures from the application programming language to get the speed they need on fetching and storing single values. Why are those operations too slow in a relational database? I consider some of the reasons below. Not all these reasons are inherent in the relational model; some have to do with architectural trade-offs made in current commercial systems, which are biased towards a data processing application mix.

(1) Each fetch or store incurs the cost of a procedure call from the application program to the database. That overhead is insignificant on a data processing transaction that accesses a field in every tuple in a relation, but is a burden when accessing a single tuple. A procedure call can't compete with simple offset addressing for accessing a field of a record in program memory. This overhead is largely a language limitation. Relational interfaces mostly don't allow packaging a sequence of DML commands in a single call to the database, much less providing more sophisticated control structures.

(2) Connections between entities in a relational system are logical, through keys. At least one address translation is required to get from a key value to the location of a tuple. In program memory, records can point to other records directly.

(3) Normalization and other encoding of complex design structures pads the levels of indirection between an entity and a subcomponent even further. Reassembling the peices of an entity in the database requires taking a join, but it is an odd type of join, as it involves

one or a few tuples from many different relations. Invoking the same machinery to compute such as join as is used for large relations takes us far away from memory access speeds. Certain useful data structures, such as arrays, just don't have any efficient encoding in record-based models. Implicit ordering information ends up being represented by explicit position numbers. Long text or byte string values are a other examples of structures that are hard to represent in record-based systems.

(4) The common strategies for transactions and recovery that work well in commercial systems are locking and logging. Both put a lot of overhead on transactions that do individual updates to tuples. Neither has been validated as the optimal approach in an environment with long transactions and data fields that may change many times before commit. An update-in-place strategy with a write-ahead log makes a lot of sense if a modified field is being changed once during a transaction. Most of the disk accesses involved with the update are moved ahead of the commit point, leaving little I/O for commit time. Few disk I/Os at commit time means high transaction throughput. Some logical logging schemes can even commit multiple transactions with one disk write. The appropriateness of such a strategy is not so clear if the field is updated many times. (Consider pushing VLSI cells around looking for a more compact layout.) In such a case, we are trading several disk accesses during the transaction body to remove one after the commit point. However, slower response time during the transaction in exchange for faster commits doesn't make much sense in a design environment. A designer would gladly accept a few second pause when saving a design for fast editing abilities.

(5) Answers are copied. The result of a relational query is a new relation, whose tuples must be composed of copies of other tuples or parts of tuples.

Many of these problems are exacerbated in the distributed workstation environments common with design projects. Commercial database systems mainly run on a single processor. Such a system would occupy one node in a local network, and be accessed through messages over the network. Crossing the application-database boundry many times becomes even more prohibitive, because it involves a remote procedure call, and another layer of copying, if the database doesn't include network communication features. Nor is it likely that the architectures of such systems could be adapted easily to a local network system, as logging and locking both access a centralized resource during transaction execution, or at least require distributed agreement.

Why Can an Object-Oriented DBMS Do it Faster?

In this discussion, I am treating object-oriented databases with behavior, that is, ones that can associate methods or operations with classes of objects. Here are reasons an object-oriented database could be made faster at fetching and storing fields than a relational system. The numbers correspond to items in the last section.

(1) Having an execution model means one message sent from the application program can do multiple field accesses in the database, at the cost of one procedure call. Design operations aren't really single fetches and stores, but typically involve following a path to another entity or filling in a new object in a class. A relational DML can't express arbitrary patterns of field access in a single operations, while OODB languages can.

(2) Objects can refer to subcomponents by identity, not state (key values). Thus, one level of mapping can be removed. However, even one level of mapping, from object identifier to main memory address, puts us more than a factor of ten from straight memory-structure speeds. There are techniques to reduce the cost of this mapping. Global object identifiers can be swizzled to local memory addresses when an object resides in main memory as in POMS [Cockshott, et al. 1984] or LOOM [Kaehler and Krasner 1983]. POMS delays the translation until the first use of a field. Thus, the first use incurs the mapping cost, but subsequent uses are at program structure speeds. Direct translation has some drawbacks, as it is difficult to move objects back out to disk on a one-by-one basis to make room for other objects. Moving an object back to disk requires finding all references to it using the local memory address. LOOM avoids this problem by having local references go through

an in-memory object table (using the reference as an index into the table) and allowing *leaf* objects that are in-memory "stubs" for objects whose state is only on disk. An alternative to external-to-internal reference replacement is maintaining a cache of main memory addresses for object identifiers. This mechanism requires a few more operations than replacement for object accesses subsequent to the first, but allows easier movement of objects back to disk, as the memory and disk formats of an object are identical.

(3) Complex design entities can be represented more directly in an OODB, with less encoding, meaning fewer levels of indirection to access one conceptual entity. In particular, many OODBs support array types directly. Some include large indexed objects with the ability to insert items in the middle, in time proportional to the size of the inserted section, rather than to the portion of the object past the insertion [Carey, et al. 1986].

(4) Long design transactions require that database users not be oblivious to each other or that the database support multiple versions of objects. These conditions, along with object identity, make optimistic concurrency control with shadowing for recovery look promising. If each application gets a "personal" copy of the database operate on, there is no need to log changes or lock items centrally during the body of a transaction. (Of course, a transaction must keep track of objects that it touched for use by the commit protocol, but it can do that locally.) Also, fewer disk I/Os are required while the transaction runs (at the expense of more I/Os at commit). In particular, an object can be updated many times without writing to disk or making a log entry each time. Object storage schemes that use an object table are especially amenable to shadowing [Maier, et al. 1986].

(5) Answers to queries can often be constructed by collecting references to objects in the database, rather than by copying the objects.

I believe all these advantages can combine to let OODBs handle individual object accesses fast enough to keep design data under database control while being manipulated by tools. Actually, I expect they can give better performance than the current method of reading in a file or some relations and building structures in program memory. First, they avoid batch load. A designer doesn't have to wait until a tool loads in an entire design to begin work. Also, if the database system, rather than virtual memory, handles buffering, it should be able to achieve better hit ratios, as it knows more of the semantics of the data.

Many of the OODBs being developed also seem well suited to mapping onto a workstation/server architecture. Most behavioral OODBs are internally structured into a storage layer and an execution layer. The storage layer handles data management functions, and the execution layer provides a workspace for evaluating methods or operations. There is one instance of the storage layer, but an instance of the execution layer for each application session. It seems reasonable to move the execution layer up to the workstation, leaving the storage layer on a central storage server [Rubenstein, et al. 1987]. For such a split to work out, the communication between the two layers has to be minimized. Shadowing and optimistic concurrency control help because there are not constant calls to the storage server during a transaction, as with logging and locking. For data fetching and storing, after some initial conversation, an execution layer can read and write pages directly across the network to move objects between workspace and disk. This arrangement is preferable to having the storage layer do the mapping between pages and objects, since the latter makes the storage layer a bottleneck for concurrent transactions. Even functions that must be centralized in the storage server, such as allocating free pages and new object identifiers, can be batched to cut down on communication overhead. An application session can be be given pages and identifiers in large chunks, only returning to the storage server when the chunk is exhausted. One question that remains for a local network architecture is how associative access support should work. If set queries are processed on the storage server, then an execution layer must push all its changed state down in order to have such a query evaluated in the middle of a transaction. Perhaps it can just push down the identifiers of modified objects, and the storage layer can request the state of only those it needs to process the query.

Research Questions

There are still numerous problems to be solved in learning how to optimize OODB performance for design applicatons. I discuss a few here.

(1) What new tradeoffs exist in an architecture with one more level in the memory hierarchy (disk to central server over a network to local workstation memory to processor cache)? What is the major bottleneck? Disk bandwidth? Server CPU? Network communication? Does it make sense to have execution capabilities in both the workstation and the storage server, or is the expense of maintaining state consistency too much? To what uses can disks on the local workstation be put? Recovery and checkpointing? Scheme or method cache?

(2) Different design tools have different access patterns. Thus, any single clustering scheme for objects into pages is unlikely to suit all of them. A design goes through phases: initial creation, design rule checking, correction, extraction, simulation. Should the objects in a design be reclustered in each of these phases? If so, how? Batch reclustering? Adaptive reclustering based on currently active access paths? The latter course has the disadvantage of making the design data well clustered for the *last* thing you did, not the current task.

(3) Most OO languages manufacture the storage layout of instances from the class definition. In a language with encapsulation, there exists the freedom for tuning the internal representation. Even if the basic building blocks for classes are sets, arrays and records, there are multiple ways to lay out combinations of them on secondary storage. An array of records might be contiguous records in order, or it could be a list of offsets to the records. We need to look at mechanisms for specifying representations easily. We also may want to support different representations on disk and in memory. For example, depending on the tool or method in use, we may want to load just a particular fragment of the state of an object.

(4) Optimization and access planning techniques need to be extended for the types of operations that show up in design applications, such as single-object manipulation and traversals of logical object structure, for example, connectivity checking or fault propagation in electrical CAD. Also, updates are common during early parts of the design cycle, and we must figure how to optimize them as well.

(5) The flexibility of dynamic binding of messages to methods is useful during system development. It means the decision of the exact typing of the instance variables for a class can be delayed. Such flexibility is desirable in modeling a domain for the first time. Later, when the types of variables become apparent, we want to declare them, in order to do more static binding and get better performance. However, we want to be able to return to the dynamic binding case if the system is undergoing modification. The general problem is keeping track of binding environments, when they change, and what has been bound relative to a certain state of an environment. The goal is to figure out when an environment is changing slowly enough that binding relative to it is worthwhile. The range of variability runs from slowly changing, such as the connection of class names to class definitions, and of types to instance variables, to more quickly changing, such as the association of messages to methods, values to instance variables and objects identifiers to memory locations. But even in the most rapidly changing environments, binding may pay off. An object may reside only briefly in main memory, but it could still be worthwhile to map references to its identifier to references to its memory location, if the object is expected to be accessed several times before leaving memory.

(6) It seems too early to settle on a single semantics of versions and configurations. Still, we can start to identify features that should be present in the database kernel to support a range of version semantics. Some such features might be cheap copies of large objects, support for compound identifiers and resolution of such identifiers, and template objects, to be used for representing the common structure across multiple configurations [Maier

1987].

(7) There is a tension between encapsulation and maintaining auxiliary access paths. Databases conventionally index on structure, but to respect encapsulation, an OODB should index on the results of an operation. That is, the operation is applied to all elements of a set, and the results of the applications are organized into an index. The problem is knowing, for an arbitrary operation, that it returns the same value when applied twice in a row and which other operations on an object can cause the result of the indexed operation to change. The current choices are to violate encapsulation, and index on structure, or trust the implementor of a class to ensure that certain operations behave properly for indexing. The latter choice runs counter to what is provided in relational systems, where index maintenance is guaranteed, no matter how a relation is defined.

Acknowledgements

My thanks to Arne Berre, Rod Butters, Craig Brandis, Goetz Graefe and Jacob Stein for letting me try some of these ideas out on them. They can take credit for the good ones and disown the losers.

The paper is an expansion of a position paper that appeared in the 1986 Workshop on Object-Oriented Database Systems [Maier 1986].

References

T.M. Atwood, "An object-oriented DBMS for design support applications," Proceedings IEEE COMPINT 85, 1985.

J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, H.-J. Kim, "Data model issues for object-oriented applications," ACM TOOIS 5:1, January 1987.

M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E. Richardson, E.J. Shekita, M. Muralikrishna, "The architecture of the EXODUS extensible DBMS: a preliminary report," Univ. of Wisconsin TR 644, May 1986.

W.P. Cockshott, M.P. Atkinson, K.J. Chisholm, P.J. Bailey, R. Morrison, "Persistent object management system," Software--Practice and Experience, 14, pp49-71, 1984.

K. R. Dittrich, W. Gotthard, P. C. Lockemann, "DAMOKLES—A database system for software engineering environments," Proc. IFIP Workshop on Advanced Programming Environments, Trondheim, June 1986, Lecture Notes in Computer Science, Springer-Verlag.

R. H. Katz, E. Chang, "Managing change in a computer-aided design database," VLDB XIII, September 1987.

T. Kaehler, G. Krasner, "LOOM—large object-oriented memory for Smalltalk-80 systems, in *Smalltalk-80: Bits of History, Words of Advice,* G. Krasner, ed., Addison-Wesley 1983.

G. S. Landis, "Design eveolution and history in an object-oriented CAD/CAM database," 31st IEEE COMPCON, March 1986.

D. Maier, "Why database languages are a bad idea," Workshop on Database Programming Languages, Roscoff, France, September 1987.

D. Maier, "Why object-oriented database systems can succeed where others have failed," Proceedings of the International Conference on Object-Oriented Databases, September 1986.

Maier, D., J. Stein, A. Otis, A. Purdy, " Development of an Object-Oriented DBMS", ACM, Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR, September, 1986.

W. B. Rubenstein, M. S. Kubicar, R. G. G. Cattell, "Benchmarking simple database operations," Proceedings ACM-SIGMOD International Conference on Management of Data, May 1987.

Wegner, P. and S. Zdonik, "Language and Methodology for Object-Oriented Database Environments", Proceedings of the Nineteenth Annual International Conference on System Sciences, Honolulu, Hawaii, January, 1986.