Type-Safe Heterogeneous Sharing can be Fast

B. Liskov, A. Adya, M. Castro, Q. Zondervan
Laboratory for Computer Science,
Massachusetts Institute of Technology,
Cambridge, MA 02139

Abstract

Safe sharing is a desirable feature of an object oriented database because it protects valuable database objects from program errors in application code. It is especially desirable in a heterogeneous environment in which applications are written in various programming languages, many of which have unsafe features.

However, safe sharing is not without its potential performance costs. This paper explores these costs. It describes a number of techniques that improve performance without sacrificing safety, and presents results of experiments that evaluate their effectiveness. The results show that some of these techniques are very promising, allowing safe sharing to be achieved with essentially no performance penalty.

Keywords: Object-oriented databases, object-oriented languages, type-safe languages, heterogeneity, performance.

1 Introduction

Type-safe sharing is a desirable attribute for an object-oriented database since it means valuable database objects are less likely to be damaged by errors in application code that uses the database. It is even more desirable in a heterogeneous environment in which applications are written in various programming languages, many of which have unsafe features.

Type-safe sharing means that database objects are manipulated only by calling their methods. This provides the benefits of abstraction and modularity. Abstraction allows users to conceptualize objects at a higher level, in terms of their methods, and to reason about objects behaviorally, using their specifications rather than their implementations. Modularity allows local reasoning about correctness, by just examining the code that implements the abstraction, with the assurance that no other code can interfere. These properties have proved very useful in programming, especially for large programs. We believe that they are at least as important for object-oriented databases, where the set of programs that can potentially interact with objects in the database is always changing; type-safe sharing can ensure that none of this new code can cause previously written code to stop working correctly.

Both abstraction and modularity work only if backed up by an encapsulation mechanism that ensures that only the code that implements an object's abstraction has access to the representation of the object. The only practical way to ensure encapsulation is to limit the programming language used to implement database objects to be *type-safe*. Such a type-safe programming language must provide a mechanism for implementing data abstractions that limits access to object representations, and it must forgo various unsafe features such as unsafe casts, explicit memory management, and unchecked array indexing.

Although the language used to implement database objects must support safe sharing, it is desirable to allow applications to be written in whatever language is most congenial to the application programmer, e.g., C or C++. Many of these languages are not type-safe, and therefore application code written in them must not be allowed

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136. M. Castro is supported by a PRAXIS XXI fellowship.

to run inside the database. Instead the application must run outside the database, and it must interact with database objects only by calling their methods (rather than manipulating them directly). Note that we require here both that the code run in response to the method calls be safe, and also that it be the right code for that object and method.

Support for safe sharing is not without its potential performance costs, and other systems have chosen to forgo safety for improved performance. Many systems, e.g., [6, 12, 5] allow the application code to directly manipulate database objects; ODMG [7] also follows this approach because it allows methods of shared objects to be written in various unsafe programming languages. O2 [10] and GemStone [16] store methods in the database, which means that it is possible to guarantee that the right code runs in response to application calls. However, the languages provided by O2 for method definition are not safe (for example, one of these languages is an extension of C). GemStone does better since programmers use a variant of Smalltalk to define the methods stored in the database, but GemStone exports an unsafe interface to client applications that allows direct access to an object's internal state.

This paper explores the performance penalty of supporting safe sharing. It describes techniques that improve performance without sacrificing safety. It also presents the results of experiments that indicate the effectiveness of the various techniques. Our results show that although some approaches are very slow, others provide excellent performance. They indicate that it is not necessary to abandon safe sharing to achieve good performance in a heterogeneous environment; instead you can have both. The experiments were done in Thor, a new object-oriented database system that supports safe sharing, and its type-safe language, Theta, but the results are applicable to other systems and languages.

This paper is organized as follows. We begin in Section 2 by briefly describing the context for our work. Section 3 describes the techniques that can be used to achieve safe sharing. Section 4 presents the results of our performance experiments. We conclude in Section 5 with a summary of what we have accomplished.

2 The Thor System Interface

Thor is a new object-oriented database system intended for use in heterogeneous distributed systems [11]. It provides highly-reliable and highly-available storage so that persistent objects are likely to be accessible when needed in spite of failures. It supports heterogeneity at the machine, network, operating system, and especially programming language levels. Thor makes it easy for programs written in different programming languages to share objects. Different client languages might be used for different applications, or even for components of the same application. Furthermore, even when client code is written in unsafe languages (such as C or C++), Thor guarantees the integrity of the persistent store.

Thor provides its users with a universe of objects. Each object in the universe is encapsulated: it has a state that is not visible to users, and can be accessed only by calling the object's methods. Each object also has a type that determines the signatures of its methods.

Applications can make use of Thor objects by starting up a *session* with Thor. Within a session an application carries out a sequence of transactions; our current approach starts up a new transaction each time the client terminates the previous one. Each transaction consists of one or more method calls. Clients can terminate a transaction by requesting a commit or abort. In the case of a commit, the system may not be able to commit the transaction, e.g., because the client has made use of stale data, and in this case the transaction aborts. If the commit succeeds, we guarantee that the transaction is serialized with respect to all other transactions, and that all its modifications to the persistent universe are recorded reliably [1].

Method calls return either *values* or *handles*. A value is an integer, boolean, character, or real. A handle is a short-lived pointer to a Thor object. A handle is valid only for the current client session; an attempt to use it in a different session will result in an error.

Figure 1 illustrates the Thor interface. Note that Thor objects remain inside Thor, and the code for object methods is stored in Thor and

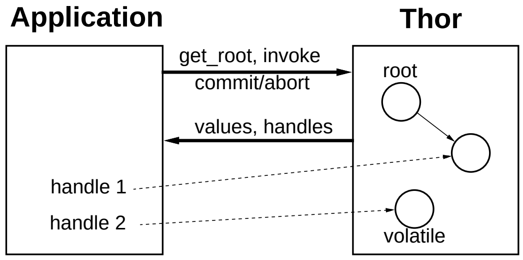

Figure 1: The Thor Interface

runs inside Thor; this is an important way in which Thor differs from other systems. Code is implemented using a new programming language called Theta [9, 14]. Applications are implemented in a programming language augmented by a *veneer* that makes it easy for the application to interact with Thor.

Theta is a strongly-typed language that guarantees that objects can be used *only* by calling their methods. In addition, all built-in Theta types do run-time checks to prevent errors, e.g., array methods do bounds checking. Theta is based on a heap with automatic storage management. It distinguishes between specifications (which are used to define the interface and behavior of a new type) and implementations (code to realize the behavior). It provides support for both parametric and subtype polymorphism, and it separates code inheritance from the subtyping mechanism. More information about Theta can be found in [9, 14].

All type definitions and implementations are stored in Thor. The type definitions constitute a *schema library* that can be used for browsing, for compilation of Theta programs, and for producing programming language veneers as discussed next.

A *veneer* [2] is a small extension to a programming language that makes it easy for programs written in that language to use Thor. A veneer provides procedures that can be called to open a session or commit a transaction, and it provides translations between scalars stored in Thor (e.g., integers) and related types in the application language. It also provides a way of interacting with (non-scalar) Thor objects. This is accomplished by a *stub generator* for that language. A stub generator is a program that reads Theta type definitions and produces a *stub type* in the application language together with *stub operations* that correspond to methods of the Theta type. We refer to objects belonging to these stub types as *stub objects*. Stub objects are created in response to calls to Thor; when a call returns a handle, the application program receives a stub object containing the handle. When a stub operation is called on a stub object, it calls the corresponding Thor method on the object denoted by the handle in the stub object, waits for a reply, and returns to the caller.

When Thor receives a call from the veneer, it checks whether the handle is valid, whether the object has the method being called, and whether the call has the right number and types of arguments. This *dynamic type checking* is necessary for unsafe languages, since stub code can be corrupted or bypassed. If the client language were safe, checking would not be necessary, which would speed up the interaction between the application code and Thor.

We have defined veneers for C, C++, Perl, and Tcl, although only the C++ veneer will be provided in Thor0, the first Thor release. Experience shows that defining a new veneer is not very difficult. It is not necessary to modify the language compiler. Also veneers can be easily provided regardless of whether the application language provides support for objects. More information about veneers can be found in [2, 15].

3 Safe Sharing Techniques

The simplest way of achieving safe sharing is by keeping the application and the database in separate protection domains, which might run on the same or on different machines; we will refer to this as the *all-outside* approach. Each method call invoked by the application is then a cross-domain call. If the call performs a considerable amount of work (e.g., a complex query on a large set of objects), the domain-crossing costs are relatively unimportant. However, if the call does relatively little work, the domain-crossing costs dominate and can result in poor application performance. This scenario is the worst case for safe sharing and in the rest of this paper, we will analyze its costs along with techniques to reduce them.

The execution time of an application's transaction can be explained by the following model. Suppose the client invokes N methods on database objects. Each of these calls has an overhead of S seconds; S is the average cost of communicating the call from the client to the database, type checking the call at the database, and communicating the result from the database back to the client. (Thus S includes the cost of marshaling/unmarshaling arguments and results.) Also, suppose running the N calls requires X pairs of domain crossings, where each domain-crossing pair has an average cost of C seconds, i.e., C is the cost of a domain crossing round trip. Finally, let the remaining cost be R; R corresponds to the cost of running the transaction without cross-domain calls. Then the total elapsed time T will be:

$$\mathsf{T} = \mathsf{X} * \mathsf{C} + \mathsf{N} * \mathsf{S} + \mathsf{R} \qquad \text{(EQN)}$$

Note that the useful work done in the above equation is R; the rest of the time is the penalty paid for safe sharing. Note also that R includes concurrency control and persistency-related costs that are incurred while the application computation runs; however it does not included the cost of committing (or aborting) the application transaction, since we want to focus on the cost of the computation itself. R also includes safety-related costs incurred by Theta (such as array bounds checking).

Below we discuss techniques that reduce T. There are three techniques — *batching*, *code transfer*, and *sandboxing*.

3.1 Batching

Batching is a technique that reduces the total execution time by reducing the number of domain crossings. Rather than having a crossing for each call, calls are grouped into batches, and an entire batch is sent to the database in one crossing. Thus, X is divided by the average batch size.

Earlier work [2] investigated one way of doing batching, called *batched futures*. In this approach, whenever a method call returns a handle, the call is batched for future execution; this makes sense because the only thing that can be done with a handle is to make a subsequent call that uses the corresponding object. A batch is sent to the database when the application makes a call that requires an immediate response, such as a call that returns a value. Although this approach does improve performance (over an approach that has one domain crossing per call), experiments with OO7 [3] showed that the average batch size was low (3.27), and this limited the amount of speed up.

This led us to investigate a second approach, *batched control structures*[18], in which batches corresponding to entire loops can be constructed. This approach gives much higher batching factors, and thus reduces X substantially. It also reduces S. A batch describes a loop by containing a description of each call in a single iteration of the loop. It is type-checked when it is received by the database, and the type-checking cost is proportional to the size of the batch, rather than the number of calls that will occur when the batch runs in the database. Similarly, the marshaling and unmarshaling costs are also proportional to the size of the batch.

Both batched futures and batched control structures incur a higher cost C for each domain crossing. This extra cost is relatively unimportant for batched control structures since batch sizes are very large, but it can have a significant impact for batched futures.

3.2 Code Transfer

Code transfer moves a portion of the application into the database. The application then makes calls on the transferred code; thus code transfer effectively increases the granularity of the calls made by the application. Typically, the entire application computation is not moved into the database since there are certain operations that do not need the database, e.g. user interface operations. However, there is some computation/navigation that is done between such non-database operations; this computation can be captured in a procedure and transferred to the database. Queries are a code transfer technique; here we are interested in other kinds of code transfers that cannot be expressed in a query language such as OQL or extended SQL.

Clearly, we cannot just take a procedure written in an unsafe language like C++ and move it into the database. We need some safe ways of performing code transfer. Here are a few possibilities:

1. Write the procedure in the database language (Theta in the case of Thor). The only problem with this approach is that it requires the application programmer to know the database language.

2. Translate a piece of the application code into the database language. The piece being translated should correspond to a procedure that does not have any free variables, i.e., all communication with the environment is via arguments and results (and reads and writes of database objects). In addition, the language used to write such procedures would be limited to a simple (safe) subset of the application language. This approach has the disadvantage of requiring safe subsets of various application languages to be defined; a more serious disadvantage is that a compiler is needed for each application language.

3. Translate the application code to Java [13] byte codes. As in the previous approach, we assume that a procedure without free variables is being translated. It is possible that translators from various application languages to Java will be common in the future, and therefore this approach avoids the problem of needing translators to a non-standard language like Theta. A potential disadvantage is that Java byte codes are interpreted, which is unlikely to give performance competitive with the above two approaches. However, this problem can be overcome by compiling the Java byte codes.

These approaches reduce cost by reducing the number of calls N, which also reduces the number of domain crossings X. They can result in greatly improved performance because very large reductions are possible.

Each approach requires *validation* to ensure that the code being transferred to the database is legal. For the first two cases, the code that runs inside the database is object code produced by the Theta compiler. Therefore, we need to validate that the code really is produced by a legitimate Theta compiler. This can be accomplished by running the compiler inside the database, or running the compiler outside and communicating with it over a secure, authenticated connection. Validation is part of the methodology for loading Java byte codes; it is performed by the *byte code verifier*.

Validation can be very expensive, but it is not necessary to do it on every call. Instead, code can be validated and then stored in the database for future use; this is a kind of memoizing.

The transferred code makes calls to database code and we need to ensure that these calls are type-correct. In the first two approaches this *type checking* is done when the Theta code is compiled. In the third case the byte-code verifier does most of the type checking, although a few checks are left for runtime.

3.3 Sandboxing

The final technique for reducing the cost of safe sharing is to transfer object code into the database and use a sandboxing technique [17] to provide safety. This technique basically involves putting (unsafe) application code/data in a restricted range of virtual memory addresses and then allowing the application code to access only these addresses. The application's object code is *encapsulated*, i.e., augmented with runtime checks so that each jump/store/load operates only on valid addresses. Wahbe et al. have shown that the overheads of this approach are relatively low. With this approach the number of calls N remains the same, and furthermore each call requires a domain crossing. However, the cost of a domain crossing (C) is greatly reduced.

Sandboxing (like the code transfer techniques) requires both validation and checking. Sandboxed code is validated using an object code verifier [17]. Checking must be done on every call made from the sandboxed code to the database; thus each call still incurs a cost S. Checking costs have been ignored in earlier work on sandboxing; our experiments shed light on these overheads.

4 Experiments

To evaluate the approaches to safe sharing discussed in the previous section, we ran experiments that measured their performance. Our experiments were designed to highlight the costs of the various approaches.

The experiments ran on Thor. Thor has a distributed client-server architecture. Objects are stored persistently by a set of servers. Applications access these objects by communicating with a front-end (FE) process that caches copies of persistent objects. Both application and FE run on the client machine, in separate protection domains.

The experiments ran the FE and application on a DEC 3000/400 workstation, with 128 MB of memory and OSF/1 version 3.2. The code was compiled with DEC's CXX and CC compilers with optimization flag -O2. We found that the performance of our experiments was very sensitive to the layout of code in memory (we observed differences as large as 30%). Therefore, in order to reduce the noise due to misses in the code cache, we used *cord* and *ftoc* [8], two utilities that reorder procedures in an executable by decreasing *density* (i.e. ratio of cycles spent executing the procedure to its static size). We used cord to obtain different executables optimized for each particular experiment.

The application and the FE communicate using a shared memory buffer with a simple synchronization based on spinning until a flag takes some desired value, and yielding the processor after each unsuccessful test. This gives us a very fast domain crossing. The cost of ping-ponging an integer between the client and the FE is $31\mu s$.

The experiments ran the single-user OO7 benchmark [3]. The OO7 database contains a tree of *assembly* objects, with a height of 7; each non-leaf assembly has three children. The leaves point to three *composite parts* chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic parts* linked by *connection* objects; each atomic part has 3 outgoing connections. The (*small*) database has 20 atomic parts per composite part and a total size of 7 MB. We implemented the database in Theta, following the specification of OO7 [4] closely.

We report results for traversal T1, which performs a depth-first, read-only traversal of the assembly tree and executes an operation on the composite parts referenced by the leaves of this tree. This operation is a depth-first, read-only traversal of the entire graph of a composite part.

We ran the traversal 52 times within a single transaction; we report the average elapsed time of the 50 middle runs. The standard deviation was always below 1% for values above one second and below 6% for values below one second. We used an FE cache large enough to hold the entire database.

This experimental methodology strives to minimize the costs that are not related to safe sharing. In particular, it ensures that there is no disk I/O, no message passing across the network, and no commit cost in the measured traversal execution times. Therefore, the costs reflect only the computation at the application and FE, and the communication between them.

4.1 Results

The results of our experiments are shown in Figure 2. The experiment labeled all-outside ran the entire traversal in the application. In this experiment, the application called methods on database objects that simply fetched values of instance variables. The experiment labeled batching shows the performance when using the batched futures approach. The same-process experiment gives an approximation to what can be expected from a sandboxing technique. The remaining two experiments are code transfer techniques in which part of the application was written in Theta. In all-inside, the entire traversal ran inside the FE [1]. In part-inside, the assembly tree is traversed as in all-outside, but the composite parts and their associated graphs are traversed in the FE using Theta code. The figure shows that the all-outside approach performs much worse than all-inside (approximately 40 times slower). Note that this is a worst case, both because our experimental methodology avoids network and disk I/O overheads and because the methods called by the application are very simple and perform no computation. In fact, the figure shows that when methods perform some computation the overhead is significantly reduced (part-inside is only approximately 2 times slower than all-inside).

To explain the experiments' results, we present some data that allows them to be analyzed using our equation *EQN*. The number of cross-domain calls performed in these experiments is presented

[1] This is in fact the way that the OO7 traversal is supposed to run according to [4].

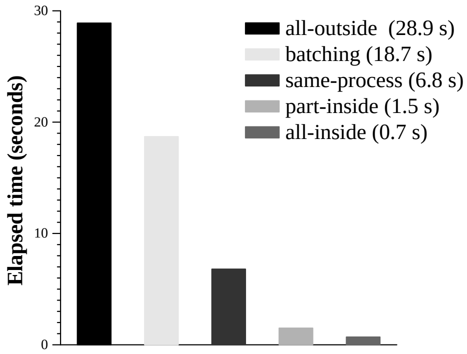

Figure 2: Performance comparison of different safe sharing techniques.

Experiment	X
all-outside	450886
batching	138020
same-process	450886
part-inside	9112
all-inside	1

Table 1: Number of cross domain calls.

in Table 1. Table 2 presents a breakdown of the execution time for the all-outside experiment; it shows that context switching is the dominant overhead in this case. The table allows us to compute the parameters of the analytic model presented in *EQN*, obtaining $C = 49\mu s$, $S = 13.5\mu s$, and $R = 0.7s$.

In Table 2, the context switching overhead was determined by comparing the elapsed time in all-outside with the elapsed time obtained by running the same code with the client and FE executing in the same process, and communicating using the same mechanism (labeled *same-process*). The safety and marshaling overhead was determined by subtracting the all-inside traversal time from the same-process time. This slightly underestimates this cost because the traversal code executes faster in C++ than in Theta as we will show. [2]

The *batching* experiment reduces the number

Context switching ($X \times C$)	22.1
Safety and Marshaling ($N \times S$)	6.1
Inside traversal (R)	0.7
Total	28.9

Table 2: Breakdown of elapsed time for the all-outside traversal (seconds).

of pairs of domain crossings X from 450886 to 138020 but it does not change the number of calls N performed by the client. Therefore, the analytical model predicts an elapsed time of $13.6s$. The difference between the predicted value and the observed value of $18.7s$ is due to the overheads introduced by the machinery to handle batched futures. This technique has the advantage that it is transparent to the programmer and portable. However, the experiment shows that its effectiveness is limited by the factor of reduction in the number of domain crossings (i.e. the size of batches) and the overhead introduced by the machinery to handle batching.

The same-process experiment represents an approximation to the cost of running the traversal using sandboxing. In this experiment, the application and FE execute in the same process. The traversal still makes $N = 450886$ calls to database object methods; each of these requires a *fault* domain crossing, but we have assumed that these crossings have zero cost ($C = 0$). We also neglect the overheads of running sandboxed client code; according to [17] sandboxed client code runs approximately 20% slower. Therefore we are only left with the cost S for making and type checking the calls. As expected performance improves noticeably over the all-outside case, but it is still significantly worse than running the entire traversal in Theta; this happens because sandboxing does not reduce the safety and marshaling/unmarshaling costs, i.e. $N \times S$, which can be significant for large N. [3]

We do not show any experiments that give the performance of the batched control structures

[2] It is interesting to note that a significant portion of the safety cost, $2.6s$, is due to a reference counting scheme that transparently garbage collects unused object handles. We have designed and implemented a scheme that eliminates this cost but forces the client to free object handles explicitly.

[3] The figure shows sandboxing to be 9.4 times worse than all-inside, but this experiment was done with garbage collection of handles; the same experiment done with explicit freeing of handles performs 5.8 times worse than all-inside.

mechanism, since it is not running in the current version of Thor. The part-inside gives an idea of what its performance might be (i.e., we could batch a loop for traversing each composite part). However, batched control structures would not perform as well as part-inside because it incurs additional overhead (the system translates the batch into a parse tree that is then interpreted to run the batch).

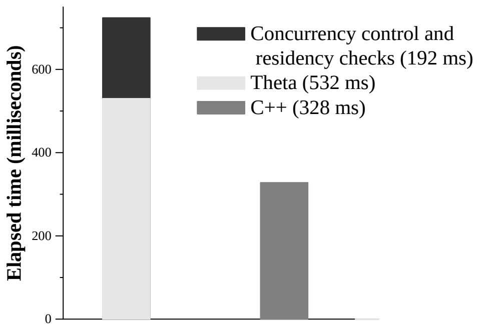

Figure 3: Cost of providing safety in Theta.

The results in Figure 2 show that code transfer techniques that run part of the application in the FE using Theta have the best performance. The all-inside value represents a best case for these techniques. To put this all-inside value in perspective we compare it with the elapsed time measured when running the same traversal using a C++ program that mimics the Theta code. The result of this comparison is presented in Figure 3. The C++ program does not incur any overhead for concurrency control and residency checks, but of course it ought to. Therefore, we break the all-inside execution time into overhead that is related to concurrency control and residency checks, and the actual cost of running the traversal in Theta. We conclude that Theta is 62% slower than the corresponding C++ implementation that provides no safety guarantees.

Table 3 presents a more detailed breakdown of the execution time for the all-inside traversal. The array bounds cost is necessary for safe sharing since it prevents violations of encapsulation. The remaining cost is not necessary for safe sharing. Here we are generating and checking exceptions for such things as integer overflow. Although safe sharing is still possible without these checks, it is worth noting that overall system safety is improved by having them: they will prevent erroneous transactions from committing that might commit in their absence. Furthermore, we have already designed a scheme that will reduce a large part of this cost.

Concurrency control	120
Residency checks	72
Exception generation/handling	156
Array bounds checking	36
Traversal time	340
Total	724

Table 3: Breakdown of elapsed time for the all-inside traversal (milliseconds).

Thus the only cost in the all-inside case that is intrinsic to safe sharing is the array bounds checking. This introduces an overhead of approximately 11% relative to the observed elapsed time for the C++ code. These results show that it is not very expensive to provide safe sharing of objects using a type-safe statically typed language like Theta. Furthermore, our Theta compiler is an experimental prototype, and therefore it does not generate highly-tuned code. We expect overheads to be reduced by using common compiler optimization techniques such as code motion.

5 Conclusions

Safe sharing is a desirable feature of an object-oriented database because it protects valuable database objects from program errors in application code. It is especially desirable in a heterogeneous environment in which applications are written in various programming languages, many of which have unsafe features.

However, safe sharing is not without its potential performance costs. This paper has explored these costs. It describes three techniques — batching, code transfer, and sandboxing, that improve performance without sacrificing safety, and presents results of experiments that evaluate their effectiveness. The results show that code transfer techniques are especially promising, allowing safe sharing with almost no performance penalty.

Code transfer techniques provide good performance because they reduce domain-crossing costs by reducing the number of domain crossings, and allow the checking of most method calls to happen prior to runtime, either when the code is compiled, or (in the case of Java byte codes) when it is verified.

The problem with code transfer techniques is that they require either an application programmer to write code in the unfamiliar database language, or they require compilers from arbitrary unsafe languages to the database language. The one exception here is translations to Java byte codes; this approach is promising because we may expect these translators to come into existence for other reasons (e.g., Web applets). Although running an interpreter for the byte codes inside the database is unlikely to produce performance comparable to what can be achieved when translating directly into the database language, this problem can be overcome by compiling the byte codes. This way we should be able to achieve the best possible performance (equivalent to the all-inside experiment).

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[2] P. Bogle and B. Liskov. Reducing cross-domain call overhead using batched futures. In *OOPSLA '94*, 1994.

[3] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, DC, May 1993.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. Technical Report; Revised Version dated 7/21/1994 1140, University of Wisconsin-Madison, 1994. WWW users: see URL ftp://ftp.cs.wisc.edu/OO7.

[5] M. J. Carey et al. The EXODUS extensible DBMS project: An overview. In *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann, 1990.

[6] M. J. Carey et al. Shoring up persistent applications. In *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, May 1994.

[7] R. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1994.

[8] Digital Equipment Company. *OSF/1 Manual Page*.

[9] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of OOPSLA '95*, Austin TX, October 1995. At `ftp://ftp.pmg.lcs.mit.edu/pub/-thor/where-clauses.ps.gz`.

[10] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[11] B. Liskov et al. The language-independent interface of the Thor persistent object system. In *Object-Oriented Multi-Database Systems*, pages 570–588. Prentice Hall, 1996.

[12] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine*. Addison-Wesley, Englewood Cliffs, NJ, May 1996.

[14] B. Liskov et al. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Lab for Computer Science, Cambridge, MA, February 1994. Available at `http://www.pmg.lcs.mit.edu/papers/thetaref/`.

[15] B. Liskov et al. Safe and efficient sharing of persistent objects in thor. In *Proceedings of SIGMOD '96*, 1996.

[16] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[17] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, Asheville, North Carolina, USA, December 1993.

[18] Q. Y. Zondervan. Increasing cross-domain call batching using promises and batched control structures. Technical Report MIT/LCS/TR-658, Laboratory for Computer Science, MIT, Cambridge, MA, June 1995.