Benchmarking persistent programming languages: quantifying the language/database interface

Antony L. Hosking
hosking@cs.purdue.edu
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

September 28, 1995

1 Introduction

A motivating factor in the development of object-oriented databases is that they reduce the *impedance mismatch* [Copeland and Maier 1984] between programming languages and database systems. One benefit is that a straightforward mapping between the type system of the programming language and that of the database reduces the conceptual barrier for the developer's of database applications. Object-oriented databases also represent an opportunity for more efficient implementation of the language/database interface, since it is possible for the type system of the language to make guarantees about the safety of programs executing against the database. A type-safe program can freely manipulate permanent data without fear of corrupting the database – the type system guarantees it. Thus, there is no need for an unwieldy interface between program and database for the manipulation of data. So long as mechanisms exist to make database objects directly available to the program in memory, the program can access and modify that data directly, without making calls to the underlying database system. In effect, data are cached by the application program for efficient manipulation.

Persistent programming languages [Atkinson et al. 1982; Atkinson, Chisholm, Cockshott, and Marshall 1983; Atkinson, Bailey, Chisholm, Cockshott, and Morrison 1983; Atkinson and Buneman 1987] epitomize this ideal by viewing the database as a stable, persistent, extension of volatile memory, in which data may be dynamically allocated, but which persists from one program invocation to the next. The language allows traversal and modification of the persistent data structures *transparently*, without explicit calls to read and write the data. Rather, the language implementation and run-time system contrive to make persistent data resident in memory on demand, much as non-resident pages are automatically made resident by a paged virtual memory system. Moreover, a persistent program can modify persistent data and commit the modifications so that their effects are permanently recorded in persistent storage.

Atkinson, Bailey, Chisholm, Cockshott, and Morrison [1983] characterize persistence as "an orthogonal property of data, independent of data type and the way in which data is manipulated". This particular characterization has important ramifications for the design of persistent programming languages, since it encourages the view that a language can be extended to support persistence with minimal disturbance of its existing syntax.

The notion of persistent storage as a stable extension of the dynamic allocation heap allows a uniform and transparent treatment of both transient and persistent data, with persistence being orthogonal to the way in which data is defined, allocated, and manipulated. This characterization of persistence allows us to identify the fundamental mechanisms that any such persistent system must support, as the basis for any study of the *performance* of persistent systems.

To be widely accepted, orthogonal persistence must exhibit sufficiently good performance to justify its inclusion as an important feature of any good programming language. Ideally, such persistence ought not to require any unusual support from the underlying hardware or operating system, so that persistent programming can be extended to the widest possible community.

2 Essential persistence mechanisms

As in traditional database systems, a persistent system must cache frequently-accessed data values in memory for efficient manipulation. Because memory is a relatively scarce resource, it is likely that there will be much more persistent data than can be cached at once. Thus, the persistent system must arrange to make resident just those persistent values needed by the program for execution. Without knowing in advance which data is needed, the system must decide dynamically when to retrieve needed data from secondary storage into memory (although any advance knowledge that *is* available should be used to guide prefetching). Updates can be made in place, in memory, but ultimately must be propagated back to stable storage. Thus, a persistent system must provide mechanisms for the detection and handling of references to persistent data and for the propagation of any modified data back to stable storage. Efficient implementation of these mechanisms is the key to implementing a high-performance persistent programming language, since they provide the fundamental database functionality of retrieval and update.

2.1 The read barrier: object faulting

The first mechanism mediates retrieval of data from stable storage into memory for manipulation by the program. Any operation that directly accesses a data value whose residency is in doubt must first check that the value is available in memory. Such residency checks constitute a *read barrier* to any operation that accesses persistent data: before the operation can read (or write) the data it must first make sure it is resident.

2.2 The write barrier: detecting and logging updates

The second key persistence mechanism ensures that updates to cached (i.e., volatile) persistent data are reflected in the database. Making updates permanent means propagating them to stable storage. Thus, every operation that modifies persistent data requires some immediate or subsequent action to commit the modification to disk. A persistent system might write the modifications straight through to disk on every update, but this is likely to be very expensive if updates are frequent or otherwise incur very little overhead. Instead, updating the stable store is typically held over to some later time, usually at the instigation of the programmer through the invocation of a *transaction commit* or a *checkpoint* primitive operation. Either way, any operation that modifies persistent data in memory must arrange for the update to be made permanent, directly by writing the modified data to disk, or indirectly by asking the system to remember that the update was made. Recording updates in this way constitutes a *write barrier* that must be imposed on every operation that modifies persistent data; writes require additional overhead to record the update.

Implementation of the write barrier is intimately tied up in assurances of database *resilience* in the face of system failures. When a user commits to a set of updates they want to be certain that all of the updates are permanently reflected in the database. Unfortunately, a crash results in the loss of the volatile part of the database (the cached persistent data), including all updates to persistent data that have not yet been propagated to stable storage. Recovering from such a failure involves restoring the database to some consistent state from which processing can resume. Any implementation of the write barrier must take into account the mechanism by which programmers specify database consistency and the information that must be generated for recovery.

3 Fine-grained persistence and the performance problem

Conventional non-persistent programming languages, such as those in the Algol family (including Pascal, C, Modula-2, and their object-oriented cousins C++, Modula-3 and even Smalltalk) have a fine-grained view of data — they provide fundamental data types and operations that correspond very closely to the ubiquitous primitive types and operations supported by all machines based on the von Neumann model of computation. Such a close correspondence means that many operations supported in the language can be implemented directly with as little as one instruction of the target machine.

Orthogonality mandates that even data values as fine-grained as a single byte (the smallest value typically addressable on current machines) may persist independently. Clearly, this situation is significantly different from that of relational database systems, where the unit of persistence is the record, usually consisting of many tens, if not hundreds, of bytes. Where a relational system can spend hundreds or thousands of instructions implementing relational operators, an Algol-like persistent programming language must take an approach to persistence that does not swamp otherwise low-overhead and frequently executed operations.

Thus, implementations of the read and write barrier for such languages must be sufficiently lightweight as to represent only marginal overhead to frequently executed operations on fine-grained persistent data.

4 Benchmarking persistent programming languages

As described above, a persistent system embodies an extremely tight integration of programming language and database. Such systems will succeed or fail as much on their performance as programming languages as on their performance as database systems. Thus, benchmarking a persistent system requires a holistic approach that focuses on both language and run-time behavior as well as underlying database functionality. Performance metrics should include fine-grained (i.e., instruction-level) overheads in addition to the coarse-grained features typically measured by database benchmarks (e.g., transaction throughput, commit latency, disk accesses, etc.). Examples of the former include the instruction-level overheads for the read and write barriers elucidated above.

As a case study for discussion, we offer the following brief description of the performance evaluation of a persistent Smalltalk implementation [Hosking et al. 1993; Hosking and Moss 1993a; Hosking and Moss 1993b; Hosking 1995]. We present performance results for alternative implementations of the read and write barrier, running the OO1 Traversal benchmark [Cattell and Skeen 1992] and its Update analogue [White and DeWitt 1992], respectively, against the "small" OO1 database. The benchmarks were run on a 40MHz SPARCstation 2 (unified instruction and data cache). All figures quoted are significant to within a 90% confidence interval.

Table 1: Traversal: key cold/warm/hot results

Scheme	Cold	Warm	Hot				
	elapsed time (s)	elapsed time (s)	elapsed time (s)	I refs	I misses	R misses	W misses
non-persistent	0.056	0.056	0.0547	1016219	6983	9109	50
ID, lazy	2.3	0.11	0.066	1303593	7083	10584	323
ID, opportunistic	2.5	0.11	0.056	1039126	6823	9361	271
FB, opportunistic	2.5	0.12	0.0553	1027172	6866	9445	251
ID, eager	3.9	0.13	0.055	1016224	6771	8969	230
FB, eager	3.9	0.14	0.055	1016230	6785	8957	234
trap, eager	4.7	0.16	0.055	N/A	N/A	N/A	N/A

I refs = instruction references; I misses = instruction cache misses
R misses = data cache read misses; W misses = data cache write misses

4.1 The read barrier: object faulting

The object faulting results compare the following alternative implementations of the read barrier:

- non-persistent: non-persistent Smalltalk, with the database entirely resident
- ID: tagged persistent identifiers
- FB: pointers to tagged resident proxies (fault blocks)
- trap: pointers to page-protected resident proxies

In addition, the eagerness to swizzle is varied (where applicable):

- lazy: obtain a direct pointer only when traversing the reference (source locations are not updated)
- opportunistic: intra-segment references are swizzled when a segment of objects is made resident
- eager: all references to a target object are swizzled when the target is first made resident

The results (Table 1) illustrate a clear tradeoff as the system warms up, with the up-front overheads of swizzling paying off only for warm and hot iterations. Hot performance reveals the payoff to be obtained through swizzling, with significantly reduced overheads (as highlighted by the number of instructions executed per iteration), and performance very close to that of non-persistent.

4.2 The write barrier: detecting and logging updates

The write barrier results compare the following alternative implementations:

- objects: on update set bit in object header; on checkpoint scan for changed objects
- remsets: on update enter object reference into a hash table data structure called a *remembered set* (cf. [Ungar 1984; Ungar 1987]); on checkpoint iterate over set
- cards-n ($16 \leq n = 4^k \leq 4096$ bytes): on update set bit in dirty card table; on checkpoint process objects and fragments in dirty cards
- pages (4096 bytes): same as cards-4096 but driven by virtual memory page protection traps on first write to a page

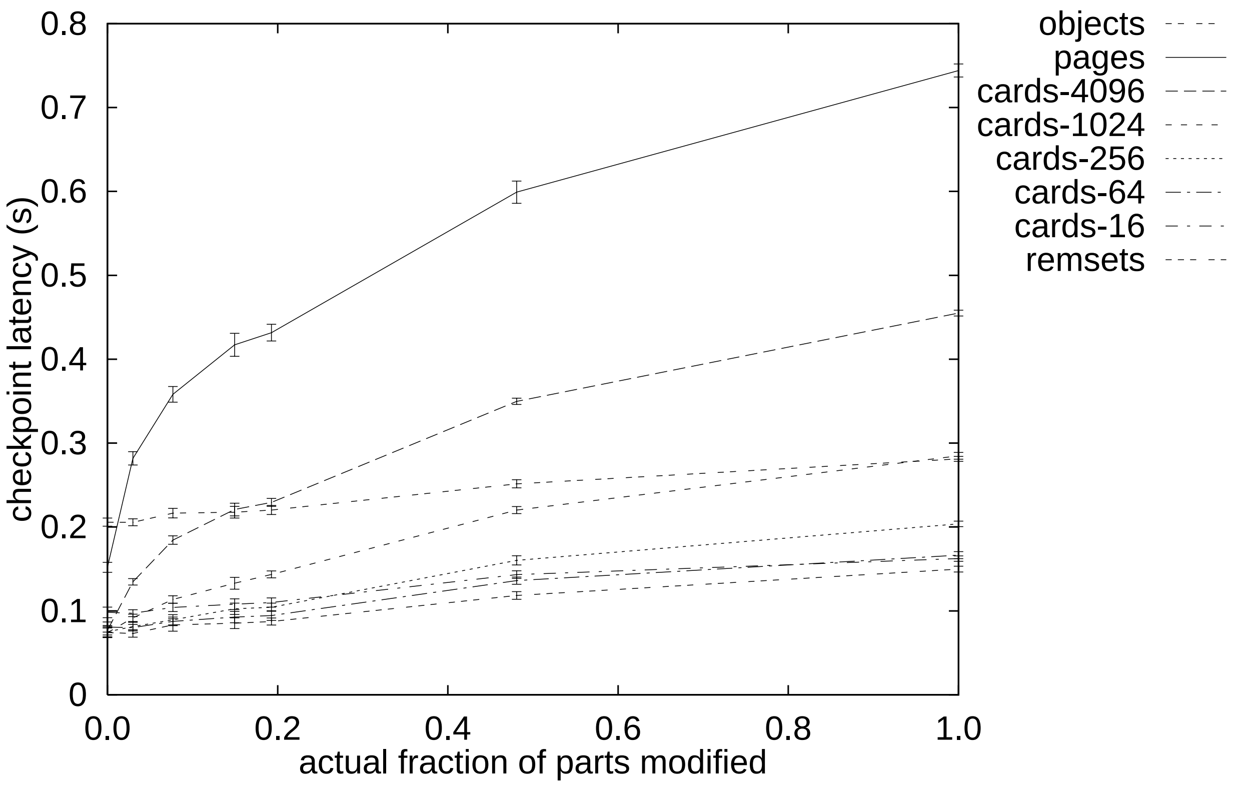

Figure 1: Update: checkpoint latency

All schemes write only object differences to the log, for minimal log volume. We report hot results for long-running transactions so as to focus on the intrinsic costs of each scheme.

The first set of results (Figure 1) concern the latency of checkpoints. The factor that has most impact on checkpoint latency is the granularity of the information recorded by the update mechanism, since that determines the amount of data that must be compared to generate the difference log. The pages scheme is most expensive, both because of the large granularity it implies, but also because of the overhead to reset page protections on checkpoint – the system call to do this has very high cost. The cards schemes are essentially ordered by granularity, with 16-byte cards having performance almost as good as remsets – the difference is mostly due to the overhead of scanning the card table versus the compactly-represented remembered set.

Run-time overheads come into play only when transactions are long enough for computation to dominate checkpoint overhead. The intrinsic run-time overheads of each scheme are reported in Table 2 (for details as to the derivation of these results see [Hosking 1995]). Note that the results for pages have been adjusted to account for an anomaly in the hardware cache behavior present in the raw results for that scheme (the unified cache led to contention between one data location and one instruction). Given these results and the earlier checkpoint latencies it is straightforward to calculate the minimum number of Update traversals that must be performed per checkpoint before the trap-based pages implementation outperforms the equivalent software-driven cards-4096 implementation. Note how the frequency/density of update affects the tradeoff: amortization of the up-front per-checkpoint overheads occurs with fewer Update traversals for higher update probabilitis. Moreover, pages is preferable only in extreme cases, when transactions are particularly long or updates extremely frequent and dense.

Table 2: Long-running Update: run-time overheads

	per part modified		Net (per update)	
Scheme	Time (cycles)	Instructions	Time (cycles)	Instructions
non-persistent	$159\pm_{90\%}0$	$81\pm_{90\%}0$	$0\pm_{90\%}0$	$0\pm_{90\%}0$
scan	$159\pm_{90\%}1$	$81\pm_{90\%}0$	$0\pm_{90\%}1$	$0\pm_{90\%}0$
objects	$173\pm_{90\%}1$	$89\pm_{90\%}0$	$7\pm_{90\%}1$	$4\pm_{90\%}0$
remsets	$249\pm_{90\%}4$	$139\pm_{90\%}1$	$45\pm_{90\%}2$	$29\pm_{90\%}1$
cards-16	$188\pm_{90\%}2$	$89\pm_{90\%}0$	$15\pm_{90\%}1$	$4\pm_{90\%}0$
cards-64	$185\pm_{90\%}1$	$89\pm_{90\%}0$	$13\pm_{90\%}1$	$4\pm_{90\%}0$
cards-256	$183\pm_{90\%}2$	$87\pm_{90\%}0$	$12\pm_{90\%}1$	$3\pm_{90\%}0$
cards-1024	$184\pm_{90\%}1$	$89\pm_{90\%}0$	$13\pm_{90\%}1$	$4\pm_{90\%}0$
cards-4096	$185\pm_{90\%}2$	$89\pm_{90\%}0$	$13\pm_{90\%}1$	$4\pm_{90\%}0$
pages (raw)	$219\pm_{90\%}1$	$81\pm_{90\%}0$	$30\pm_{90\%}1$	$0\pm_{90\%}0$
pages (adjusted)	$169\pm_{90\%}1$	$81\pm_{90\%}0$	$5\pm_{90\%}1$	$0\pm_{90\%}0$

$$\text{net overhead per update} = \frac{\text{overhead per part} - \textsf{scan}\text{ overhead per part}}{2}$$

Table 3: Long-running Update: break-even points for cards-4096 versus pages

update probability	checkpoint latency (seconds) pages (p)	cards-4096 (c)	parts modified per traversal (m)	break-even point (traversals, t)
0.00	$0.152\pm_{90\%}0.006$	$0.077\pm_{90\%}0.006$	0	∞
0.05	$0.282\pm_{90\%}0.008$	$0.135\pm_{90\%}0.004$	98	$2308\pm_{90\%}366$
0.10	$0.358\pm_{90\%}0.009$	$0.184\pm_{90\%}0.005$	252	$1062\pm_{90\%}167$
0.15	$0.42\pm_{90\%}0.01$	$0.221\pm_{90\%}0.008$	490	$625\pm_{90\%}105$
0.20	$0.43\pm_{90\%}0.01$	$0.229\pm_{90\%}0.005$	632	$489\pm_{90\%}74$
0.50	$0.60\pm_{90\%}0.01$	$0.350\pm_{90\%}0.004$	1577	$244\pm_{90\%}32$
1.00	$0.744\pm_{90\%}0.008$	$0.455\pm_{90\%}0.003$	3280	$136\pm_{90\%}16$

$$t = \frac{p - c}{\frac{13\text{ cycles per update}}{40\text{MHz}} \times m \times 2\text{ updates per traversal}}$$

5 Conclusions

We argue that benchmarking persistent programming languages requires consideration of both coarse- and fine-grained performance metrics, including instruction-level measurements typical within the programming language community, as well as measurements of database functionality. In particular, it is useful to explore the difference in performance between non-persistent languages and their orthogonally persistent counterparts, since that measurement is likely to be one of the deciding factors in the acceptance of persistent programming languages. Moreover, benchmarks for object-oriented databases should be amenable to these sorts of measurements.

References

ATKINSON, M., CHISOLM, K., AND COCKSHOTT, P. 1982. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices 17,* 7 (July), 24–31.

ATKINSON, M. P., BAILEY, P. J., CHISHOLM, K. J., COCKSHOTT, P. W., AND MORRISON, R. 1983. An approach to persistent programming. *The Computer Journal 26,* 4 (Nov.), 360–365.

ATKINSON, M. P. AND BUNEMAN, O. P. 1987. Types and persistence in database programming languages. *ACM Computing Surveys 19,* 2 (June), 105–190.

ATKINSON, M. P., CHISHOLM, K. J., COCKSHOTT, W. P., AND MARSHALL, R. M. 1983. Algorithms for a persistent heap. *Software: Practice and Experience 13,* 7 (March), 259–271.

CATTELL, R. G. G. AND SKEEN, J. 1992. Object operations benchmark. *ACM Transactions on Database Systems 17,* 1 (March), 1–31.

COPELAND, G. AND MAIER, D. 1984. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, pp. 316–325. *ACM SIGMOD Record 14,* 2 (1984).

HOSKING, A. L. 1995. Lightweight support for fine-grained persistence on stock hardware. Ph. D. thesis, University of Massachusetts at Amherst, MA 01003.

HOSKING, A. L., BROWN, E., AND MOSS, J. E. B. 1993. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, pp. 429–440. Morgan Kaufmann.

HOSKING, A. L. AND MOSS, J. E. B. 1993a. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, pp. 288–303.

HOSKING, A. L. AND MOSS, J. E. B. 1993b. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, pp. 106–119. *ACM Operating Systems Review 27,* 5 (Dec. 1993).

UNGAR, D. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, pp. 157–167. *ACM SIGPLAN Notices 19,* 5 (May 1984).

UNGAR, D. M. 1987. *The Design and Evaluation of a High Performance Smalltalk System.* ACM Distinguished Dissertations. The MIT Press, Cambridge, MA. Ph.D. Dissertation, University of California at Berkeley, February 1986.

WHITE, S. J. AND DEWITT, D. J. 1992. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, Vancouver, Canada, pp. 419–431. Morgan Kaufmann.