Entity Bean A, B, C's: Enterprise Java Beans Commit Options[1] and Caching

Paul Brebner and Shuping Ran

Software Architectures and Component Technologies Group
CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra, AUSTRALIA
{Paul.Brebner, Shuping.Ran}@cmis.csiro.au

Abstract. Entity Beans provide both data persistence and the possibility of caching objects and data in the middle-tier. The EJB 1.1 specification has three commit options which determine how EJBs are cached across transactions: Option C pools objects without identity; Option B caches objects with identity; Option A caches objects and data. This paper explores the impact on performance of these different commit options, pool and cache sizes on a realistic application using the Borland Application Server.

1 Introduction

Entity Beans are part of the Enterprise Java Beans (EJB) 1.1. specification [1] and provide a data persistence mechanism using a relational database. In conjunction with Container Managed Persistence (CMP) the Entity bean lifecycle also allows for caching in the EJB container. The provision for caching in the EJB model is consistent with previous research which suggests that caching is a critical requirement for the performance of object-oriented systems in general [2, 3], and the performance and scalability of distributed object systems such as Corba [4, 5], and Java/EJB [6, 7].

Three commit options (A, B, C) determine what is cached across transactions: objects without identity (C); objects with identity (B); objects with data (A). However, the specification does not mandate support for all three options. Vendors can choose to implement one or more commit options. Some vendors have chosen to support all three options to distinguish their products from others (e.g. Borland Application Server [8]), or only two options (WebSphere), while other vendors either don't support commit options explicitly (E.g. WebLogic) or only implement option C (the simplest) on the basis that object creation is cheap relative to database access, and object caching doesn't result in any performance improvement (e.g. SilverStream [9], iPlanet).

While conducting performance evaluations of a number of EJB products we experimented with various deployment settings for Entity beans. We varied commit options, and pool and cache sizes to determine their impact on performance, and to find optimal settings. During the course of these experiments we discovered a

[1] EJB Entity Beans can have one of three commit options: A, B and C.

number of interesting features of the interaction between the products, application, commit options, pool and cache sizes, and the way the tests were run. This paper reports on results for Borland Application Server (BAS), version 4.5.0[2].

The following sections explain in more depth the Entity bean lifecycle, and how it enables object pooling, and object and data caching. *Stock-OnLine* is the test application used [10], and the overall requirements are described, along with the Entity bean implementation and the test setup used. Finally, the four experiments performed are described and the results presented and analysed.

2 Entity Bean Lifecycle and Caching

2.1 Lifecycle and Persistence

The main purpose of Container Managed Persistence (CMP) Entity beans is to provide automatic persistence for their data state in a relational database. This simplifies programming and CMP Entity beans are portable across all the databases supported by EJB 1.1. compliant containers.

Entity beans can be in one of three states: Does not exist; Pooled; Ready (Figure 1).

Instances in the pool are not associated with any particular object - they don't have a primary key. Any pooled instance may be used to execute the entity bean's finder methods.

Pooled instances move to the Ready state when they are Created or Activated, but only if another object with the same identity is not already in the ready state. Once in the Ready state an instance has a particular object identity (it has a primary key), and Load and Store (to synchronise the data state with the database) and Business methods may be called on it.

A Ready instance can be moved back to the Pooled state by:

- Passivation: Disassociating the instance from the object identity (so it has no primary key), possibly even during a transaction, or
- Removal: Removing the entity object including the object's representation in the underlying database.

In theory instances can be activated and passivated on demand, in which case the data state is also refreshed (Load) or saved (Store). After the *unsetEntityContext* event objects are in the "does not exist" state and may be garbage collected.

Note that even though we talk about pooled instances, the container is not required to actually maintain a pool of instances - this is an implementation detail.

[2] Ideally the experiments would be repeated on a number of different products. However, not all products explicitly support all three commit options, and other significant differences such as whether pool and cache sizes are per bean or per container make comparisons difficult.

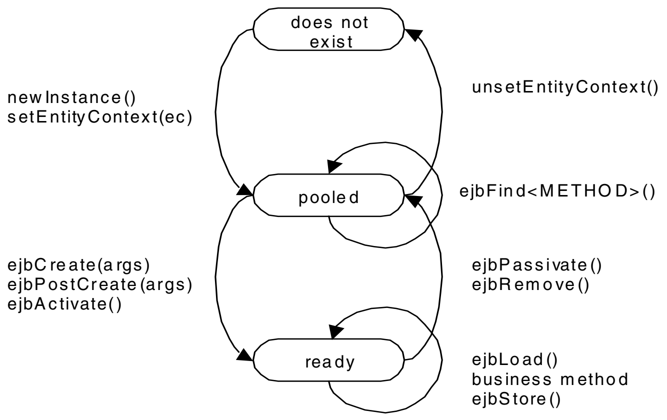

Fig. 1. Life Cycle of an Entity Bean instance (From Figure 23, Section 9.1.4. EJB 1.1. Specification)

2.2 Lifecycle and Caching

Entity beans can therefore do more than just automatically persist data. Having the Ready and Pooled states allows for *caching of Entity beans* (with and without identity) and also *caching of data* in the middle-tier.

There are three "commit options" (what the container does at transaction commit-time) known as A, B and C. These control object and data state across transaction boundaries:

- Object (*Ready*) state: Commit Options A and B maintain the object in the Ready state (activated, with primary key) across transaction boundaries. Option C doesn't, but returns it to the Pooled state (Passivation).
- Data (*Instance*) state: Options B and C do not maintain data state across transactions, but assume that the database may have been changed by another application (*shared*). They load the data state at the start of every transaction (Obviously required for Option C as there is no object in the ready state to cache the data state). Option A assumes the container has *exclusive* access to the database, allowing it to cache a ready instance with data across transactions. That is, while there is a Ready instance the database is not read again. This means that if the database is changed the cached data state will be out of synchronisation.

Option A caches data and object state, Option B caches only object state, and Option C does neither but must get/put instances from/to the Pool at the start/end of every transaction. For all the commit options the data is always written back to the database (Store) at the end of transactions (unless the container has options such as "read only" fields, or checks to see if data has actually changed before writing).

Products such as Borland Application Server that support all three commit options allow the impact of object pooling and caching, and data state caching on the performance of an experimental application to be explored. The *Stock-OnLine* application and the CMP Entity bean implementation of it will now be described.

3 The Sample Application: Stock-OnLine

3.1 Requirements

Stock-OnLine [10] is a simulation of a simple on-line stockbroking system. It enables subscribers to buy and sell stock, inquire about the up-to-date prices of particular stocks, and get a holding statement detailing the stocks they currently own. From a subscriber's perspective, the following services are offered:

- **Create Account:** A person wishing to enrol with Stock-OnLine can create themselves a subscriber account with the service provider.
- **Update Account:** A subscriber can modify their allocated credit limit.
- **Query Stock Value:** A subscriber can query the current price for a given stock. A unique identifier code, or a mnemonic code can be used to identify the stock value to be retrieved.
- **Buy Stock:** A subscriber can place an order to buy a given number of a specified stock. If successful, a transaction record is created for later processing.
- **Sell Stock:** A subscriber can place a request to sell a specified number of any stock item they have purchased through the Stock-OnLine. If successful, a transaction record is created for later processing.
- **Get Holding Statement:** A subscriber can request a statement of all the stock they have purchased through Stock-OnLine and still retain ownership of.

3.2 Database Design

In a real implementation of an on-line stockbroking system, the database would need to store many details in order to track customers, their transactions, payments, and so on. In the example used for performance measurements a simple database design has been used that contains the minimum tables and fields to allow the system to operate sensibly.

The *SubAccount* table holds basic information on a subscriber to the system, and has 4 fields. The primary key for *SubAccount* is the subscriber's account number, *sub_accno*. When a new account is created, the system needs to allocate a new, unique account number, which is done with Oracle Sequences.

Information about each stock that a subscriber can trade through Stock-Online is held in the *StockItem* table. The primary key is *stock_id*, and the other fields represent the stock's trading code, company name, and current value and recent high and low values (a total of 6 fields).

The *StockHolding* table contains information on the amount of a given stock that a subscriber holds, and has 3 fields. The primary key is compound: *sub_accno, stock_id.*

Finally, there is the *StockTransaction* table, which contains information on each transaction that a subscriber performs. The primary key is *trans_id,* which is generated in a similar manner to new accounts using Oracle sequences. The *trans_type* is a code that represents a buy or a sell transaction. Other fields record the subscriber who performed the transaction; the stock item sold or purchased, the amount of stock involved, the price and the date of the transaction.

3.3 Transaction Business Logic

An overview of the business logic for each transaction is given below. The descriptions focus on the database activity that each transaction performs, as these are the most expensive operations. Essentially each action below represents an SQL operation. Exception cases are not described, even though these are handled in the application.

- **Create Account**
  - Get a new account key
  - Insert a new SubAccount record for the new subscriber
- **Update Account**
  - Update the subscriber's credit record in the SubAccount table
- **QueryById**
  - Use the supplied stock identifier to retrieve the current, high and low values from the StockItem table using the primary key
- **QueryByCode**
  - Use the supplied stock code identifier to retrieve the current, high and low values from the StockItem table
- **BuyStock**
  - Read the SubAccount table to retrieve the credit limit for the subscriber, and ensure they have sufficient credit to make the purchase
  - Read the StockItem table to retrieve the current price of the stock the subscriber wishes to purchase
  - If the subscriber has not purchased this stock item before, insert a new record in the StockHolding table to reflect the purchase. If they do hold this stock already, update the record that already exists in the StockHolding table.
  - Get a new transaction key
  - Insert a new record in the StockTransaction table to create a permanent record of the purchase
- **SellStock**
  - Read the StockHolding table to ensure that the subscriber has sufficient holdings of this stock to sell the specified amount
  - Read the StockItem table to retrieve the current price of the stock the subscriber wishes to sell

  - Update the StockHolding table to reflect the sale of some of this stock item by the subscriber
  - Get a new transaction key
  - Insert a new record in the StockTransaction table to create a permanent record of the sale
- **GetHolding Statement**
  - Read the StockHolding table and retrieve up to 20 StockHolding records for the subscriber

The *Buy* and *Sell* transactions are more heavyweight in their demands for database operations. *CreateAccount* and *Update* perform database modifications, but are relatively lightweight. The remaining 3 transactions are all read-only, and should therefore execute quickly.

3.4 Database Initial Population

Prior to each test run, the database is populated with initial test data. Table 1 shows the database tables and their cardinality (rows).

Table 1. Initial database population

Table	Initial Number of Records
SubAccount	3000
StockItem	3000
StockHolding	3000*10 = 30,000
StockTransaction	0

3.5 Client Test Behaviour

The driver for the test application is a multi-threaded Java program running on a separate machine over RMI-IIOP. Each client thread/process performs a number of iterations of a fixed *transaction mix*. The transaction mix represents the concept of one complete business cycle at the client side. The number of transactions of each type per iteration is shown in Table 2.

Table 2. Test transaction mix

Number	Transaction
1	CreateAccount
3	Buy
3	Sell
1	Update
15	QueryById
15	QueryByCode,
5	GetHoldingStatement

This transaction mix comprises 43 individual transactions, and is a combination of mainly read-only (81%) transactions and some update (19%) transactions. Each client performs this transaction mix 10 times, with no wait times. The accounts and items use in each transaction are chosen at random, so there is no locality of reference. The only exception is the buy and sells, which are paired so that whatever is bought is sold again in the same iteration.

3.6 Entity Bean Architecture

The Container Managed Persistence (CMP) Entity bean version of *Stock-OnLine* was designed using the standard EJB design pattern of a Session bean front-end to the Entity beans. This is to prevent the client code from interacting with the Entity beans directly, only via the Session bean business methods. A single session bean implements all seven transaction business methods. Four Entity beans, one each for the database tables, manage the database access (see Figure 2). Transactions are container managed, so when a Session bean business method is invoked a new transaction is started, and all the Entity beans called participate in that transaction context.

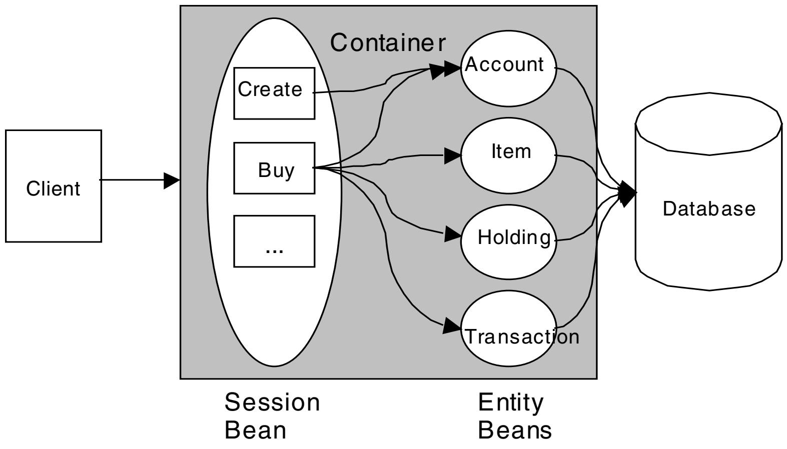

Fig. 2. Stock-OnLine Entity Bean Architecture

3.7 Entity Bean Details

Four entity beans are used to map to the four database tables. In order to understand the potential impact of the different commit options, pool and cache sizes, the dynamic behaviour of each of the four entity beans over the course of a single 100 concurrent client run is as follows (and summarized in Table 3):

- **Account** (Maps onto the SubAccount table): Initially 3000 instances. Some fields modified during tests. 1 instance created per iteration, 10 created per client. By the end of a 100 client run the number has increased by 1000 to 4000 (33% increase). 8000 instances are used (7000 read and 1000 created) during a 100 client run.
- **Item** (Maps onto the StockItem table): Initially 3000 instances. No fields modified during tests (but could be, as application allows for stock price changes). None created during tests. 36,000 instances are used (read) during a 100 client run.
- **Holding** (Maps onto the StockHolding table): Initially 30,000 instances (10 per Account). 3 instances are created per iteration, 30 created per client. Sells are paired with Buys. That is, each Holding that is bought will also be sold in the same iteration. The Holding's amount is set to 0, but the Holding is not actually deleted. The number of Holdings therefore increases constantly during test runs (at 3 times the rate of Account instances). By end of a 100 client run the number has increased by 3000 to 33,000 (10% increase). Holdings has a compound key which is sparse (i.e. key is *sub_accno* and *stock_id*, which has 9M permutations, of which only 0.33% exist initially. By end of 100 client run still only 0.36% exist). 56,000 instances are used (53,000 read, 3000 created) during a 100 client run.
- **Transaction** (Maps onto the StockTransaction table): Initially there are 0 instances, but the number of instances grows constantly. Not modified, or used during tests. 6 instances are created per iteration, 60 created per client (6 for buy/sell transactions). A 100 client run creates 6000 instances.

Table 3. Objects created and used in 100 client run

Entity Bean	Initial number of instances	New instances after 100 client run	Total instances after 100 client run	Number of instances used in 100 client run
Account	3000	1000	4000	8000
Item	3000	0	3000	36000
Holding	3000*10 = 30,000	3000	33,000	56000
Transaction	0	6000	6000	0

There is therefore one bean that grows rapidly and is heavily used during the tests (Account), one bean that is heavily used but (currently) is never created or modified (Item), one bean that has a sparse key, has 30k instances to start with and which grows and is heavily used during the tests (Holding), and one bean that is created but never used.

It makes sense to fix Transaction to be option C for these tests (otherwise it would need a large cache, but as it's never used a cache is wasted on it) and determine the impact of commit options, pool and cache sizes on the other beans.

3.8 Test Set-Up

The detailed experiments were carried out using Borland Application Server version 4.5 (BAS45), and Oracle 8.0.5, all running on NT 4.0. Three machines were used, 1 each for database, server, and client (Dual processor, 800MHz or faster, 1GB Ram; Client on an Ultra 80), with 100Mbit LAN connecting them. CPU usage on the middle-tier machine with BAS running on it was typically 90%, and CPU usage on the database machine was typically 30% or less.

Performance is measured in terms of Transactions Per Second (TPS). This is measured by dividing the total time taken by the client process (Wall-clock time) by the total number of transactions processed. For some experiments we also report the average client response times for each transaction type.

To ensure consistent and optimal results we limited the concurrency in the BAS ORB to 10 threads maximum. In BAS this limits the number of concurrent transactions, and the number of Session beans, to 10. Note that even though we repeated experiments to check results for consistency, there is still a small amount of experimental error in the TPS reported, up to about 5%.

4 Experiments

BAS calls the Pooled state pool the *Pool*, and the Ready state pool the *Cache*. Objects that have been moved out of the Pool to the "does not exist" state are called *unreferenced*, and when garbage collected they are *finalized*. We will follow this terminology.

For BAS we initially assumed that commit option B would be faster than commit option C, and used the default settings (1000 cache and 1000 pool sizes), producing reasonable results. However, we also tried option C, and surprisingly got slightly better results. We then tried to improve the performance of option B but initially found it difficult to achieve performance comparable to or better than option C. A more systematic testing approach was required.

In order to explore the impact of varying *Pool* and *Cache* sizes, we set them to different sizes determined by the total number of objects existing at the end of a run of 100 clients. A size big enough to accommodate all the starting objects and all the newly created objects corresponds to a cache hit-rate of 100% (200% is twice this). Table 4 enumerates the sizes for 10, 20, 50 and 100 percent hit-rates.

To make reporting and analysis easier we kept the cache/pool hit-rates the same for all the Entity Beans (e.g all 10%, all 20%, etc).

In theory the optimal Entity bean pool size depends *only* on the number of concurrent transactions[3] which is 10 due to limiting ORB threads. Therefore a maximum of 10 Account objects, 10 Item objects, and 200 (10 * 20) Holding objects

[3] In practice this is only the case for Commit Option C in BAS.

in the Pools are required. However, for the sake of completeness (and to test the theory) we varied the pool size across the same range as the cache.

Table 4. Example cache hit-rate sizes

Entity Bean	Total instances after 100 client run	10% cache size	20% cache size	50% cache size	100% cache size
Account	4,000	400	800	2,000	4,000
Item	3,000	300	600	1,500	3,000
Holding	33,000	3,300	6,600	16,500	33,000
Transaction	7,000	N/A	N/A	N/A	N/A

The number of objects pooled during a 100 client run is calculated from the number of instances of each object type used and the cache hit-rate. No objects are pooled with 100% cache hit-rate, and all objects are pooled with no cache (Figure 3).

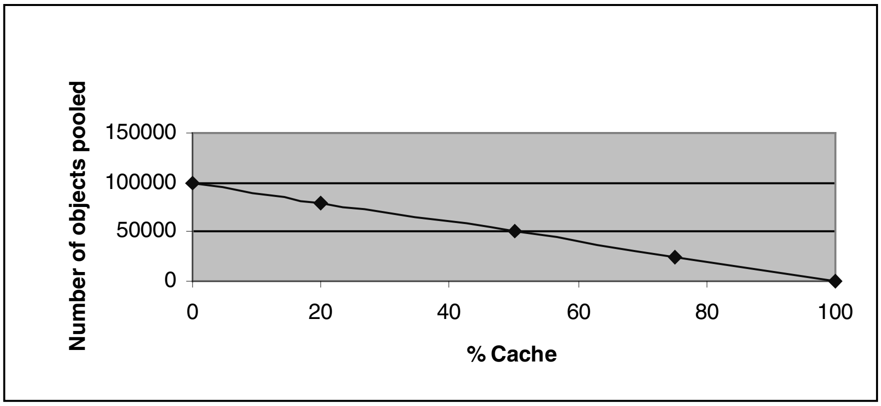

Fig. 3. Pooled objects: Expected total objects pooled during 100 client run

Assuming that object pooling has some cost associated with it we expect to see an increase in performance with increasing cache size.

For each configuration (Commit option, pool and cache sizes) we carry out the following routine:

- Restart the container/server (this clears the pool and cache)
- Load the jar file with the configuration to be tested
- Initialise the database
- Warmup the cache (using readonly operations on Account, Item, and Holding objects)
- Run 100 client test
- Record the TPS

Four experiments with BAS will be described: Experiment 1: Option C and varying Pool size (cache set to 0). Experiment 2: Option B and varying Pool size (cache set to 0). Experiment 3: Option B and varying Cache size (pool set to 20%). Experiment 4: Option A and varying Cache size (pool set to 20%).

4.1 Experiment 1: Option C and Pool

We assume that: Option C doesn't use the ready cache, so cache is set to 0. Optimal/maximum pool sizes for 10 threads are 10 Accounts, 10 Items, and 200 Holdings. We hypothesize that: 0 Pool size will be slowest. The differences in performance will be small, probably less than 10% (i.e. vendors such as SilverStream are largely correct in asserting that pooling makes no significant difference). TPS will increase up to "optimal" pool size, and then show no further increase.

Figure 4 shows the throughput (in TPS) for the different pool sizes, and Figure 5 shows the average client response times.

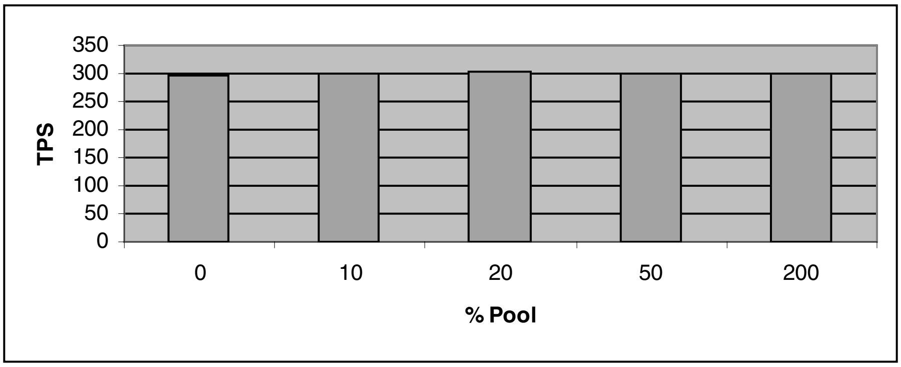

Fig. 4. Option C, Increasing Pool

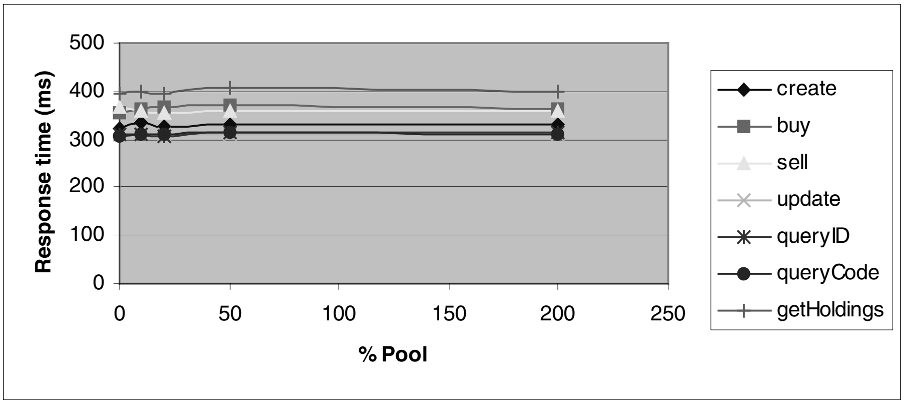

Fig. 5. Option C Client response times

Given that there is up to 5% experimental error these results do not show any significant difference in TPS for varying pool size.

With Option C we observe[4] that the total number of *ready* instances changes during the run but at any one time never exceeds the theoretical maximum based on the number of concurrent transactions (10) and objects used per transaction: i.e. 10 Account, 10 Item, and 200 Holdings.

The maximum number of *pooled* instances observed during the runs, for pool sizes greater than 0, was as expected (10/10/200) and did not depend on the actual pool size setting.

With 0 pool size the total number of unreferenced objects (in the "does not exist" state) at the end of the run was 2200 (no ready, pooled or finalised objects).

Option C with pool greater than 0 produced no unreferenced or finalised objects. That is, the objects are all being moved backwards and forwards to and from the pooled and ready states with no objects being destroyed. This is the "steady" state for the container and is an efficient use of resources: demand and supply of objects is the same, and all the objects are recycled.

Borland confirm that the pool is only resized every 5 seconds [11]. After 5 seconds any inactive beans will be discarded. In the steady state the same number of beans are being used, and they are always active. For pool sizes greater than 0 there will therefore be no (or very few) pooled objects, and less than the pool size, and therefore none need to be discarded. However, for a pool size of 0 the few inactive objects that are pooled will be be discarded, which is the observed behaviour.

The typical Option C performance is around 300TPS.

With 0 cache/pool settings, the server memory usage was only 56MB and constant. This implies that EJB applications using option C can be run on servers with relatively small amounts of memory.

Client side response times are consistent, and in order of slowest to fastest are: getHoldings, buy, sell, create, queries/update.

4.1.1 Conclusions

For BAS, using option C, a pool size greater than 0 has no significant impact on the throughput, and is probably not required. However, we observed that unreferenced objects are created, and will eventually have to be garbage collected, incurring some extra overhead. A small pool (the theoretical optimal/maximum size) is a sensible choice.

Given that 0 pool size is no slower than larger pool sizes, object creation and destruction must have minimal overhead compared to database access.

4.2 Experiment 2: Option B and Pool

This experiment is equivalent to Experiment 1, but using Option B instead. We hypothesize that: As option B with 0 cache is equivalent to Option C the results should be comparable; and different pool size will have minimal impact.

[4] BAS has good facilities for monitoring the number of instances in each state.

The results from this experiment (option B) compared with the previous experiment (option C) are show in Figure 6.

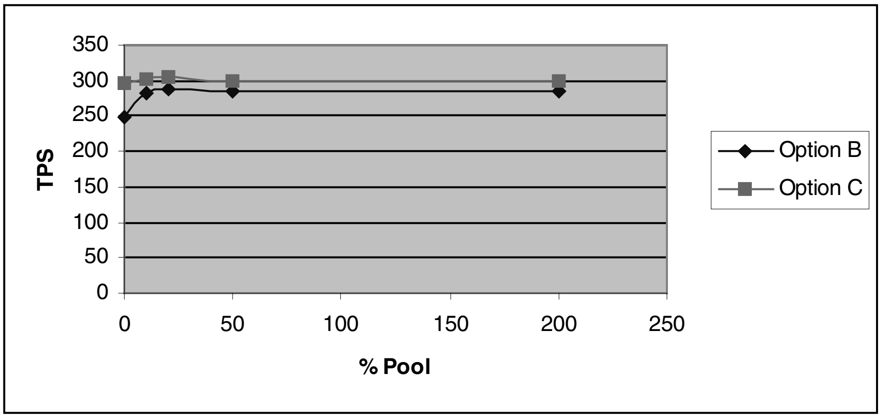

Fig. 6. Options B and C, 0 Cache, increasing Pool

0 pool size is slowest. The TPS increases by 40 from 0% to 20% pool size (16% increase), and flattens out at around 286TPS.

The best option B performance is 287 TPS, which is only 7 TPS less than the best Option C performance (304TPS). This is comparable, within experimental error. However, given that all the points from 20% cache upwards are the same, and none of them exceed the option C TPS, the indication is that the option B is really slower than option C.

Best performance with 20% pool size and above is a surprise given the theoretical expectation that the maximum pool sizes need to be only 10/10/200 (Account/Item/Holding), and the observation that only this many instances are in fact pooled using option C. We set the pool sizes to these theoretical sizes, and confirmed that this gave a slower result of 254 TPS (cf 287TPS for a larger pool).

With option C the number of ready objects never exceeded the maximum expected (10/10/200), while with option B the peak number of ready objects exceeded the maximum and fluctuated widely. This makes the "0" cache results difficult to interpret, but does seem to have an impact on the number of unreferenced objects as follows.

Even with 0 cache and 0 pool there are some ready and pooled objects at the end of the run (Table 5). The total number of unreferenced and finalised objects is in fact 100k, the expected number of pooled objects for a run. This is a lot more than the 2,200 unreferenced/finalised objects observed for Option C (0 cache/0pool).

Table 5. Option B (0 cache, 0 pool) instances at end of run

Entity bean	Ready	Pooled	Unreferenced	Finalised
Account	81	1	7500	260
Item	250	1	30000	390
Holding	700	1000	70000	700

However, with 0 cache and 20% pool the number of unreferenced/finalised objects drops significantly to about 10,000 (all of them Items). When there is no pool, discarding the objects used in a transaction incurs some overhead.

Borland indicate that locking is used to track the cache [11]. This is an overhead even with a (useless) "0" cache size and could be expected to give worse performance than for option C.

4.2.1 Conclusions

With a 0 cache size set, having at least a small pool has a bigger impact on option B than option C.

Option B (with no cache) performance is slightly slower than Option C. However, the difference is not large and we conclude they are comparable.

Even with "0" cache size set there are many ready objects during a run (more than the expected theoretical maximum as seen with option C)[5]. This results in many unreferenced objects being produced if there is no pool. Increasing the pool size reduces the number of unreferenced objects produced and correspondingly increases the TPS. We conclude that the pool needs to be large enough to cope with the peak demand for objects to and from the ready cache.

We will use a 20% pool size for the remaining experiments as a simplification to reduce the number of variables to one only - the cache size. We assume that the result for 0 cache also applies approximately to non-0 cache, but without making any explicit assumptions that this is optimal[6].

4.3 Experiment 3: Option B, Cache and Pool

We hypothesize that: Increasing the cache will increase the TPS until a maximum of 100% hit-rate; Because there is no locality of reference, random beans are accessed. Thus the working set of beans is just the number of rows in the database, and for optimal performance the cache will need to be close to 100%; For some cache size we will get better results than Option C (300TPS); The best TPS for Option B will be between 10 and 30% higher than Option C.

For this test we set the pool size to 20%. See Figure 7 for the throughput results.

Figure 8 shows average client side response times for each transaction type (3 representative points only).

During the test runs we recorded the number of instances pooled. The behaviour of the BAS container is suprising in this regard as it pools a larger number of objects than expected from the option C tests, and creates more objects nearer the middle of the % cache range (See Figure 9). The number of instances pooled is related to the peak ready pool size during the test runs. In fact BAS allows the actual ready pool size to overshoot the "maximum" setting for short periods of time, and the excess objects are then pooled reducing the ready pool size back to the "maximum" again.

[5] This is to be expected as option B pools objects with identity, so the maximum number of ready objects is the number of instances of the object (or rows in the database).

[6] An experimental check confirmed no significant difference in TPS for varying pool sizes.

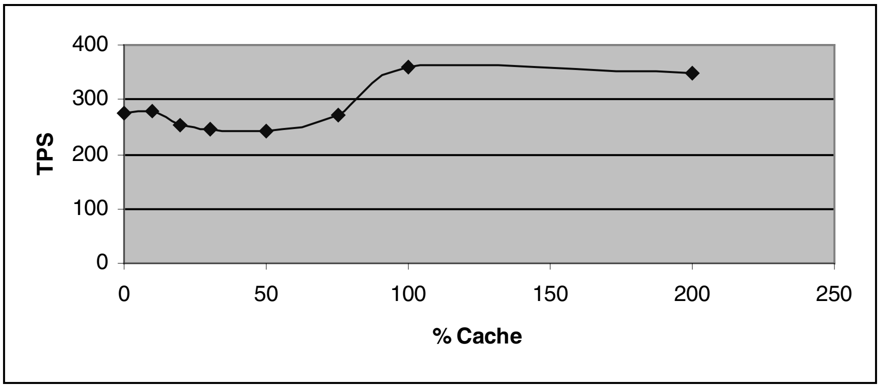

Fig. 7. Option B, Increasing cache

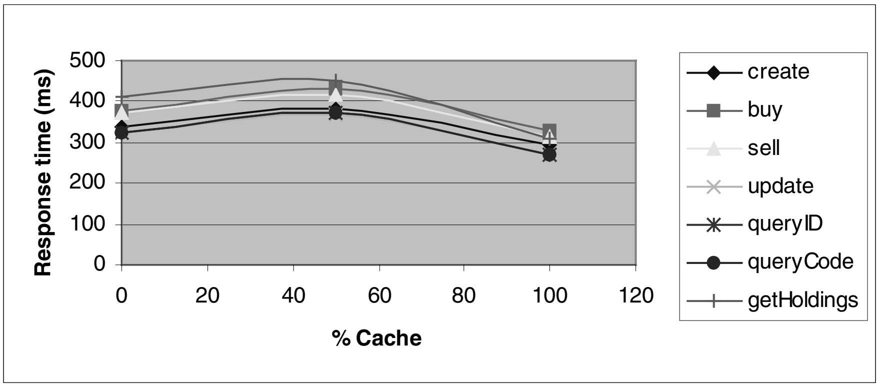

Fig. 8. Option B Client response times

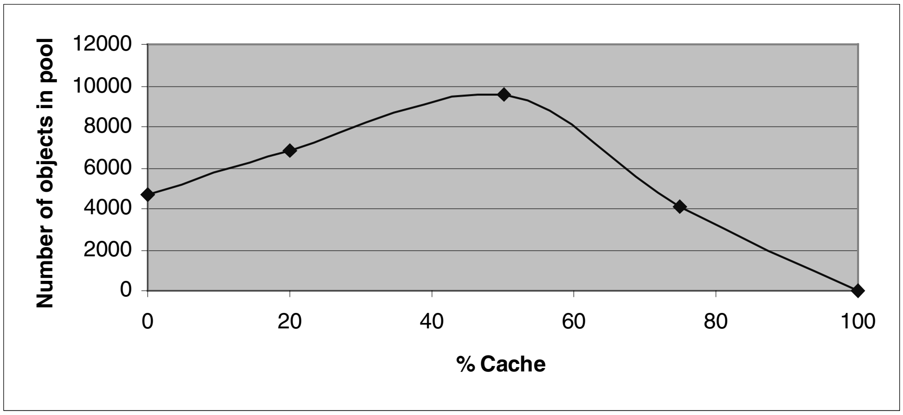

Fig. 9. Total pool sizes (account + item + holding)

With anything less than about 80% cache we get less than the best Option C throughput of 300TPS.

0 and 10% cache sizes give 277TPS, but then the TPS drops with increasing cache size up to 50%, rising to a peak of 360TPS at 100% cache size.

TPS drops again slightly to 200% (350TPS), but this is within experimental error so is unlikely to be significant.

The client response times increase, and then decrease, with getHoldings speeding up by 100% cache (to become faster than buy). Increasing the cache size has greater impact on getHoldings than the other transactions.

Initially we wondered if the number of instances in the pools is related to the shape of the TPS graph. It seemed suspicious that the peak Pool size and the minimum TPS both happen at around 50% cache size. However, we conducted further experiments with a 0 pool size which disproved this (no objects pooled during or at end of run, and same shaped TPS graph).

4.3.1 Conclusions

Option B with greater than about 80% cache is faster than Option C. 100% cache gives 360TPS which is 20% faster than option C. Option B with smaller cache (less than 80%) is slower than option C. Option B with 0 cache is close to option C performance.

The dip in TPS around 50% cache is initially suprising. However, we know that there are two competing factors, a benefit and a cost, involved in caching objects with identity:

- Benefit: More ready instances are available with increasing cache size.
- Cost: There are overheads with increasing cache size[7] (including garbage collection, retrieval of objects from the cache, managing the cache size, and pooling objects when the "maximum" size is exceeded).

It is possible to model the TPS graph by assigning suitable relative weights to these two factors, and subtracting the cost from the benefit. If this model is correct then the relative performance is just some function of the cache size benefit and cost.

Borland have given us extra insight into the impact of the cache locking overhead [11]. The smaller the cache sizes, the more locking is done as the background cache resizing thread needs to do more work to reduce the cache sizes. This provides the final key to understanding the shape of the TPS graph. The cache cost isn't linear, but consists of a locking component which *decreases* with increasing cache size, and a component which increases with increasing cache size (due to garbage collection, etc).

Because the performance is only better than option C with a large (>80% cache), for most real applications this will be difficult to achieve, particularly if there are a large number of objects, or if the number of objects grows.

Given that a significant performance improvement is only observed with a large cache, and that a large cache is practically impossible in most real applications, option C is a better option for consistently good performance and low memory usage.

[7] From the BAS ejb-cmp newsgroup.

4.4 Experiment 4: Option A, Cache and Pool

We hypothesize that: Because Option A caches data as well as object state, we expect the best Option A TPS to be substantially faster than Options B and C (In excess of 30% faster than option C); Even a small (Say 10%) cache size will give better than option C performance.

Figure 10 shows Option A results with increasing Cache, and 20% pool.

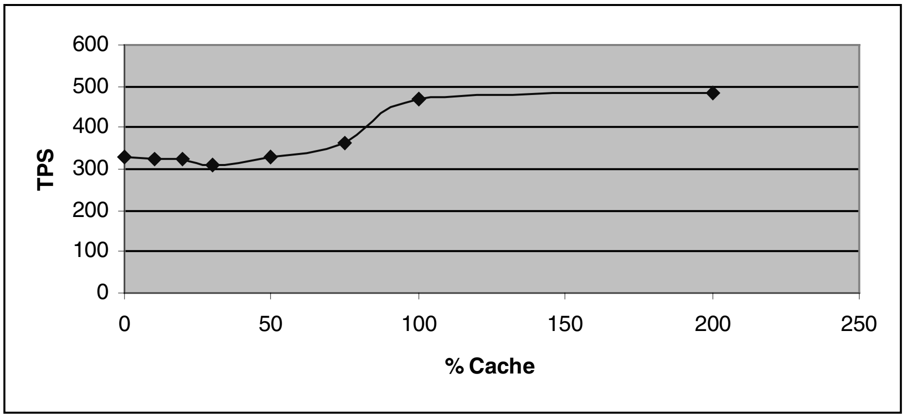

Fig. 10. Option A, Cache and Pool

We observe that: All Option A runs are faster than option C. This is because even with 0 cache specified, the container actually creates some ready instances (and therefore also caches some data). The TPS is fairly flat from 0 to 50% cache, and drops to almost the option C value at around 30% cache. From 50% cache the TPS increases, to almost the maximum at 100%.

A comparison of Options A, B, and C TPS is useful at this point to assist with comparative observations (Figure 11):

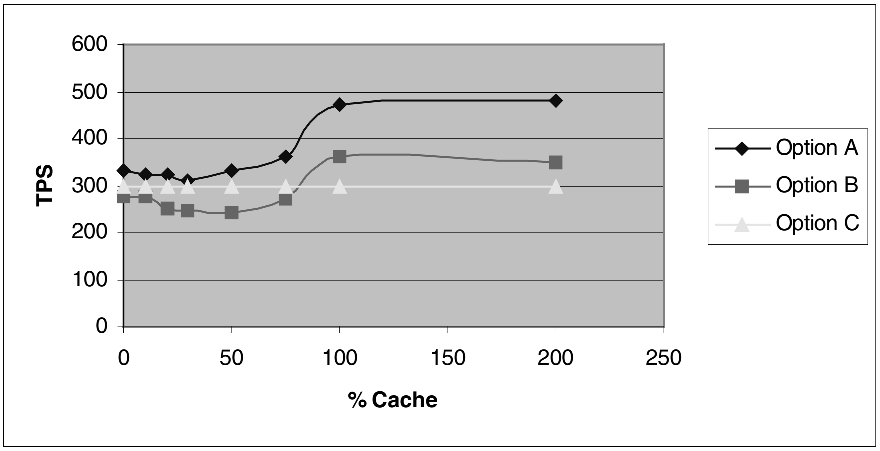

Fig. 11. Comparison of Options A, B and C

Option A has a peak TPS of 482 at 200% cache, and 471TPS at 100%. Using the 100% figure, Option A is 30% faster than the peak Option B (360TPS), and almost 60% faster than the peak option C (300TPS).

Option A and B have similar shaped graphs, although the drop in TPS between 0% and 50% isn't as bad for Option A as Option B. This is probably due to the increased efficiency of data caching almost compensating for the cost of caching in general.

In theory we would expect the Option A, B, and C 0% cache points to be the same, and the slowest, but they aren't. Previously we observed that for Option B 0% cache the container is actually creating ready instances. The maximum number of ready instances recorded for three runs with Options A, B and C (0 cache, 20% pool) are as follows (Table 6).

Table 6. Peak ready instances

Option	% Cache	Maximum Account Ready Instances	Maximum Item Ready Instances	Maximum Holding Ready Instances
C	0 (N/A)	10	10	200
B	0	240	800	1,300
A	0	260	2,700	14,000

For Option B, the ready pool size jumps around from 0 to and from the peak, and finally drops back to 0 at the end of the run. For Option A, the ready pool size increases rapidly to the peak and stays there, dropping back to 0 only at the end of the run. This proves that the cache size isn't a fixed maximum size, and that the actual number of ready instances depends on *both* the commit option and the cache size. If the actual cache size really was the fixed upper limit for the duration of the runs then the graph would look substantially different.

4.4.1 Conclusions

The benefits of the cache are more obvious for Option A than Option B. Until about 80% cache size Option B is still slower than Option C, and then only increases to a maximum just over the worst option A result. However, Option A at 75% is on par with the best Option B result, and improves even further.

5 Conclusions

The benefit of caching just object state and not data (option B) is minimal, with at least 80% cache size required to exceed option C TPS, and 100% cache size required for the maximum difference of 20% over option C. Achieving these hit-rates for cache sizes in real applications is difficult if the number of objects is large or increasing.

Option C gives consistently good results with no object or data caching. The advantage of object pooling for option C is also unobservable, and a pool size of 0 is probably adequate. Setting the pool to the maximum theoretical size (based on the maximum concurrency and the number of instances used per transaction) will however prevent any overhead caused by unreferenced objects being garbage collected. Using Option C with no pool or a minimal pool size has significant implications for hardware resources, as a BAS45 server can be run in under 64MB RAM using option C, compared to anything up to 512MB for options A and B.

Option A gives slightly better performance up to 50% cache, increasing above 50% (for this application), with substantial benefits if everything can be cached. If the EJB application is the only application accessing the database tables, then it may be worth considering using option A. The throughput is almost doubled, and is roughly equivalent to adding an extra clustered server, but without the benefits of further scalability or failover.

Finally, we stress that these results, explanations and conclusions are product, version[8], and application specific. This in itself is a major lesson. Optimising CMP Entity bean code performance is a time consuming and complex task. Using the vendor's default settings, or even using the theoretically best settings, will be unlikely to produce optimal performance for real applications. An understanding of which commit options, cache and pool sizes are supported by the vendor and what they actually do, how they relate to the EJB specification, controlled experimentation with different deployment settings, and careful observation of what the container is actually doing during the running of the application, are likely to lead to a significantly better outcome.

If CMP Entity beans are an important part of your EJB architecture then careful evaluation and choice of an EJB application server is required. One of the major areas of differentiation between Application Servers is the level of support for Entity beans, including the extent of implementation of the specification, CMP performance, tool support for Entity bean application assembly and deployment, monitoring of deployed beans, and extra features such as clustering of Entity beans. Not all EJB 1.1 compliant products are equal.

References

1. Enterprise JavaBeans Specification, v1.1. Sun Microsystems (1999), http://java.sun.com/products/ejb/docs.html
2. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R., and Barter, C.: Cache coherency and storage management in a persistent object system. In: Proceedings of the Fourth International Workshop on Persistent Object Systems. Martha's Vineyard, MA, USA (1990) 99-109
3. Brebner, P.: An Object-oriented multi-media file system for D-CART; SRS, Design and Performance analysis (Unpublished project documentation). Australian Broadcasting Corporation (ABC) TR&D, Ultimo, Sydney (1995)
4. Kordale, R., Ahamad, M., Devarakonda, M.: Object Caching in a CORBA Compliant System. Conference on Object-oriented Technologies, Toronto, Canada (1996)
5. Sandholm, T., Tai, S., Slama, D., Walshe, E.: Design of Object Caching in a CORBA OTM System. Conference on Advanced Information Systems Engineering (1999) 241-254

[8] Borland have changed some aspects of the caching and pooling implementation from BAS 4.5.0 to 4.5.1.

6. Bretl, B., Otis, A., San Soucie, M., Schuchardt, B., Venkatesh, R.: Persistent Java Objects in 3 tier Architectures. In: Atkinson. M., Jordan, M. (eds.): The Third Persistence and Java Workshop. Tiburon, California, September 1st to 3rd (1998)
7. Roman, E., Öberg, R.: The Technical Benefits of EJB and J2EE Technologies over COM+ and Windows DNA. The Middleware Company (1999)
8. Borland Application Server 4.5, http://www.borland.com/appserver
9. SilverStream Application Server, http://www.silverstream.com
10. Gorton, I.: Enterprise Transaction Processing Systems: Putting the CORBA OTS, Encina++ and OrbixOTM to Work. Addison-Wesley (2000)
11. Weedon, J.: Borland Software Corporation. Personal communication (Email), 21 May (2001)