Parallelizing SPECjbb2000 with Transactional Memory

JaeWoong Chung, Chi Cao Minh, Brian D. Carlstrom, Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{jwchung, caominh, bdc, kozyraki}@stanford.edu

1 Introduction

As chip-multiprocessors become ubiquitous, it is critical to provide architectural support for practical parallel programming. Transactional Memory (TM) [4] has the potential to simplify concurrency management by supporting parallel tasks (transactions) that appear to execute atomically and in isolation. By virtue of optimistic concurrency, transactional memory promises good parallel performance with easy-to-write, coarse-grain transactions. Furthermore, transactions can address other challenges of lock-based parallel code such as deadlock avoidance and robustness to failures.

In this paper, we use transactional memory to parallelize SPECjbb2000 [9], a popular benchmark for Java middleware. SPECjbb2000 combines in a single Java program many common features of 3-tier enterprise systems, hence it is significantly more complicated than the data structure microbenchmarks frequently used for TM research [3, 5, 8]. Since SPECjbb2000 models the operation of a wholesale company with multiple warehouses, the original code is already parallel, with a separate Java thread managing each warehouse. Different warehouses accesses mostly disjoint data structures, hence dependencies are rare. We focus on *parallelism within a single warehouse*, which is a more challenging case[1]. Conceptually, there are significant amounts of parallelism within a single warehouse as different customers order or pay for different objects. Nevertheless, all operations within a warehouse access the same data structures (B-trees), hence dependencies are possible and difficult to predict. Overall, single-warehouse SPECjbb2000 is an excellent candidate to explore the ease-of-use and performance potential of transactional memory with irregular code.

We study multiple ways of defining transactions within the SPECjbb2000 code and explore their performance potential using an execution-driven simulator for a CMP system with TM support. We demonstrate that even the simplest, coarse-grain transactions (one transaction for a whole operation) lead to significant speedups. Nevertheless, performance is still limited by frequent dependency violations on shared data structures. To mitigate the violation overhead, we use nested transactions, both closed and open, to achieve up to a 75% speedup increase. Closed nesting reduces the penalty of violations and open nesting reduces the frequency of violations. While nested transactions, particularly open, require some additional effort by the programmer, they are still significantly easier to use than fine-grain locks. Overall, we conclude that for SPECjbb2000, transactional memory lives up to its promise of good performance with simple parallel code.

2 SPECjbb2000 Overview

SPECjbb2000 is a Java benchmark, designed to measure how well systems execute Java server applications [9]. The benchmark emulates a 3-tier system for a wholesale company with multiple warehouses. Figure 1 shows the basic structure of SPECjbb2000. The first tier consists of client threads that perform random input selection to interact with the business

[1]The performance benefits from parallelizing across warehouses are orthogonal and complementary to the benefits from parallelizing within a single warehouse.

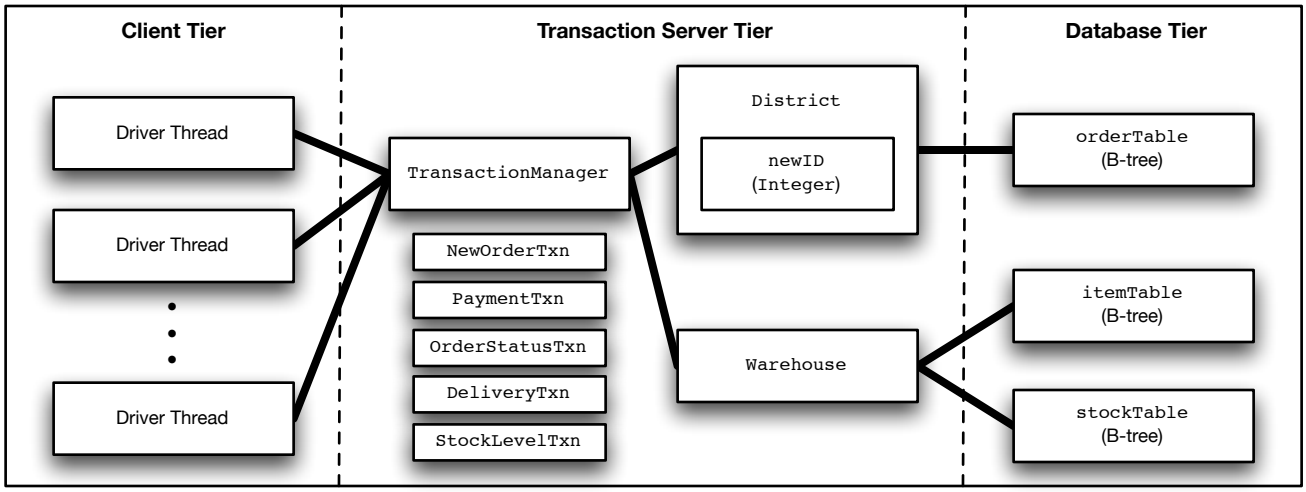

Figure 1: The major components of the 3 tiers in SPECjbb2000. The `newId` and the three B-trees objects are shared among all threads within a warehouse.

logic in the second tier. The second tier is the most significant part of the benchmark and consists of a transaction server that accesses the data managed by the third tier. SPECjbb2000 holds all data with in-memory objects using B-trees (Java objects). Since there is no database, SPECjbb2000 does no disk I/O, and since a single program includes both the server and the clients, there is no network I/O either.

The second tier can be configured to have multiple warehouses, with each serving a number of districts. By default, SPECjbb2000 runs one thread per warehouse, which exposes significant amounts of concurrency as, for the most part, warehouses operate on disjoint data. In this paper, we parallelize the second tier of SPECjbb2000 for a single warehouse and a single district. From the end user's point of view, this is an interesting case as a business should not have to introduce more physical warehouses to scale the performance of its 3-tier system. If the warehouse can scale to handle more customers, the same should be the case for the computing system that controls its operation. From the programmer's point of view, parallelizing within a single warehouse is a challenging case as the code is irregular. Conceptually, there is significant concurrency within a single warehouse, as different customers operate mostly on different objects (orders, payments, etc.). Hence, one can use multiple threads to handle these operations. However, these threads will be accessing data from the same B-trees, causing dependencies between threads. These dependencies are difficult to predict as the customer operations are randomly generated. We should point out that the performance benefits from parallelizing across warehouses are orthogonal and complementary to the benefits from parallelizing within a single warehouse.

As shown in Figure 1, the second tier contains three key modules (`TransactionManager`, `District`, and `Warehouse`). The `TransactionManager` is responsible for receiving tasks from the clients and then making the appropriate changes in `District` and `Warehouse`. To parallelize within a warehouse, multiple `TransactionManager` threads are used. The `District` module contains a global counter, called `newID`, that is used to assign a unique number to each new order. When orders are created, `District` records them in the `orderTable` B-tree. Similarly, `Warehouse` uses the `itemTable` and `stockTable` B-trees to keep track of the appropriate items. Since `newID` and the three B-trees are shared and modified by all threads, they are possible sources of dependencies and conflicts between the concurrent `TransactionManager` threads.

Figure 2 shows the sequential (baseline) pseudocode for a `TransactionManager` thread. We have abstracted out several details including thread initialization and client tasks distribution. There are five types of client tasks to process: `new_order`, `payment`, `order_status`, `delivery`, and `stock_level`. The `new_order` and `payment` types represent 43% and 43% of the total number of tasks respectively. The pseudocode focuses on `new_order` because its processing involves more updates to shared data structures. Specifically, it involves reading and incrementing the `newID`, searching for and updating nodes in `itemTable` and `stockTable`, and allocating a new entry in `orderTable`. With multiple `TransactionManager` threads operating in parallel on one warehouse, conflicts through these shared data structures will

```
TransactionManager::go() {
    while (workToDo) {
        switch (transactionType) {
            case new_order:
                // 1. initialize a new order transaction
                newOrderTx.init();

                // 2. create a unique order ID and a new order with the ID
                orderId = district.nextOrderId();   // newID++
                order = createOrder(orderId);

                // 3. retrieve items and stocks from warehouse
                warehouse = order.getSupplyWarehouse();
                item = warehouse.retrieveItem();    // itemTable.search()
                stock = warehouse.retrieveStock();  // stockTable.search()

                // 4. calculate cost and update node in stockTable
                process(item, stock);

                // 5. record the order for delivery
                district.addOrder(order);           // orderTable.insert()

                // 6. print the result of the process
                newOrderTx.display();
                break;
            case payment:
                // ...
                break;
            case order_status:
                break;
            case delivery:
                // ...
                break;
            case stock_level:
                // ...
                break;
        }
    }
}
```

Figure 2: The pseudocode for a single-threaded (baseline) SPECjbb2000 `TransactionManager`.

be frequent. With lock-based coding, a programmer could simply acquire a coarse-grain lock to protect lines 9–21. The coarse-grain lock would guarantee correct execution but would also eliminate most concurrency within a warehouse. Alternatively, and after some observation, the programmer could acquire per data structure locks for the `newID` update (line 9), the `stockTable` node update (line 18), and the `orderTable` insertion (line 21). The per data structure locks would allow additional concurrency, up to one per data structure. Finally, a programmer could implement all objects using fine-grain locks, which would allow maximum concurrency, but at maximum coding complexity. Even if the fine-grain code for the object is available in a library, the programmer must make sure that the order in which the threads access objects will not lead to deadlocks or data races.

3 Methodology

We run transaction-based code on an execution-driven simulator for a CMP that follows the TCC architecture. Details about transactional execution with TCC are available in [7]. TCC provides transactional memory using lazy conflict detection to provide non-blocking guarantees. The CMP includes 8 cores with private L1 caches (32 KBytes, 1-cycle access) and private L2 caches (256 KBytes, 12-cycle access). The read-set and write-set of the transactions generated by SPECjbb2000 fit in the processor caches, hence there is no overhead for overflows and virtualization. The processors communicate over a 16-byte, split-transaction bus. All non-memory instructions in our simulator have CPI of one, but we model all details in the memory hierarchy for loads and stores, including inter-processor communication. The simulator implements closed- and open-nested transactions as described in [6], and the simulated hardware supports up to 4 levels of nesting (or nesting depth of 4). To run SPECjbb2000, we use the Jikes RVM, version 2.3.4 [1].

Even though we performed our study on top of a particular TM architecture, our conclusions about concurrency in SPECjbb2000 and its interaction with transactional memory techniques are largely independent from implementation details. We expect that SPECjbb2000 would perform similarly on other, well-engineered, transactional memory systems.

Figure 3 presents the performance results for the various transactional versions of SPECjbb2000. The results represent execution time with 8 `TransactionManager` threads on 8 processors normalized to the sequential (baseline) execution time of a single thread on one processor. Lower bars represent better performance. The bars also indicate the percentage of execution time wasted on dependency violations between concurrent transactions. The labels above each bar represent the speedup over the baseline (maximum speedup is 8). We measured execution time by running 368 client tasks, and the results focus on benchmark execution time, skipping virtual machine startup.

4 Transactional Parallelization of SPECjbb2000

We parallelized the `TransactionManager` for SPECjbb2000 with transactions in three different ways. The first approach uses flat, coarse-grain transactions, while the other two take advantage of the hardware support for nested transactions (open and closed).

4.1 Flat Transactions

Figure 4 shows the pseudocode for the first transactional version of `TransactionManager`. This transactional code is equivalent to coarse-grain lock-based code: the whole customer task (`new_order`) is expressed as a single, atomic transaction. Similarly, each other type of task is a single, flat transaction (`payment`, `order_status`, `delivery`, and `stock_level`). This version of the code is by far the simplest for the programmer. All she has to do is to use an `atomic{}` block (2 extra lines of code) to specify that the whole task should be atomic with respect to all other threads, regardless of the number, type, or order of objects accessed within the task. In other words, the programmer is only required to understand the atomicity behavior of the application at the highest possible algorithmic level. No understanding of the exact implementation of the task or the data structures is needed.

The first bar in Figure 3 shows that the coarse-grain (flat) transactional code provides a speedup of 3.09 on the 8-processor CMP. Despite using coarse-grain transactions, there is significant concurrency as threads often operate on different types of

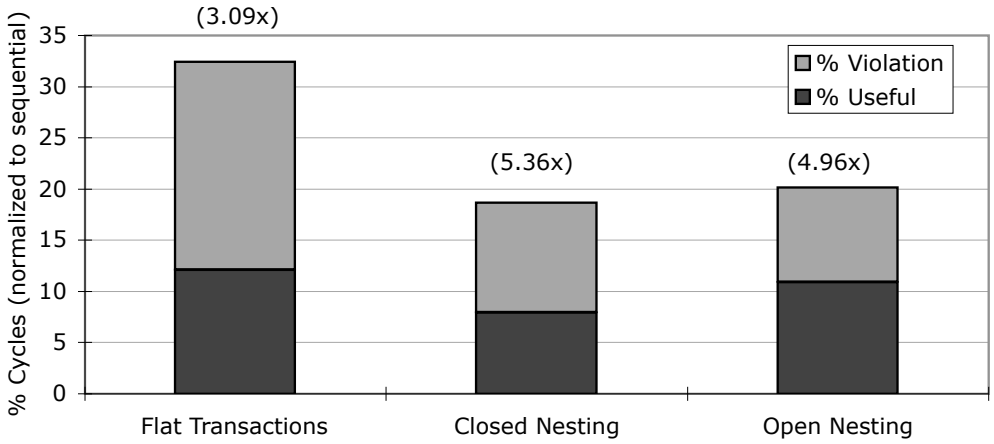

Figure 3: The execution time of the transaction-based parallel version of SPECjbb2000 for 8 processors, normalized to the sequential time. The labels above each bar represent speedups.

```
1  TransactionManager::go() {
2      while (workToDo) {
3          switch (transactionType) {
4              case new_order:
5                  atomic {          // begin transaction
6                      // 1. initialize a new order transaction
7                      // 2. create a unique order ID and a new order with the ID
8                      // 3. retrieve items and stocks from warehouse
9                      // 4. calculate cost and update warehouse
10                     // 5. record the order for delivery
11                     // 6. print the result of the process
12                 }               // commit transaction
13                 break;
14             ...
15         }
16     }
17 }
```

Figure 4: Pseudocode of the first transactional version with flat, coarse-grain transactions.

tasks, on different B-trees, or different portions of the same B-tree. Compared to coarse-grain lock-based code that gets a speedup of approximately 1, the performance of the simple, coarse-grain, transactional code is very encouraging.

Nevertheless, approximately 63% of the execution time for each thread is wasted executing transactions that eventually rollback due to a dependency violation. We used the TAPE profiling tool to identify the main sources for the violations [2]. The first main source of conflicts is the update to the `Warehouse` B-tree (Figure 2, line 18) and the `orderTable` B-tree (Figure 2, line 21). The second main source was the `newID` counter that is read-modified-written for every new order by a customer (Figure 2, line 9). We attempted to reduce the penalty and frequency of these conflicts by using nested transactions.

4.2 Closed Nesting

To reduce the overhead of each conflict between transactions from different threads, we used closed-nested transactions [6] as shown in Figure 5. Again, the whole customer task (`new_order`) is a transaction (lines 5 to 18). However, we define two nested transactions: (1) around the updates to the `newID` and the `Warehouse` B-trees (lines 8 to 12) and (2) around the updates to the `orderTable` B-tree (lines 14 to 17). The first nested transaction establishes a new order ID and identifies the items involved. The second nested transaction updates the order table. When a nested transaction begins, we track its

```
1  TransactionManager::go() {
2      while (workToDo) {
3          switch (transactionType) {
4              case new_order:
5                  atomic {   // begin outermost transaction
6                      // 1. initialize a new order transaction
7
8                      atomic {    // begin nested transaction
9                          // 2. create a unique order ID and a new order with the ID
10                         // 3. retrieve items and stocks from warehouse
11                         // 4. calculate cost and update warehouse
12                     }           // end nested transaction (merge with parent)
13
14                     atomic {    // begin nested transaction
15                         // 5. record the order for delivery
16                         // 6. print the result of the process
17                     }           // end nested transaction (merge with parent)
18                 }         // commit outermost transaction
19                 break;
20             ...
21         }
22     }
23 }
```

Figure 5: The SPECjbb2000 parallel pseudocode with closed-nested transactions.

read-set and write-set independently. Hence, if a conflict with another thread is detected and it involves only the nested (inner) transaction, we only need to rollback to the beginning of the nested transaction and not to the beginning of the outer one. For example, if we detect a conflict on the `orderTable` B-tree while the thread is in line 16, we only roll back and re-execute from line 14 as opposed to line 5. This can significantly reduce the overhead of a conflict. However, when a closed-nested transaction commits, we simply merge its read-set and write-set with the parent transaction. Hence, if we detect a violation on the `newID` counter while the thread is in line 14, we have to roll back all the way to line 5 as we can no longer separate the first nested transaction from the outermost one.

The second bar in Figure 3 shows that closed-nested transactions eliminate approximately half of the overhead from violations and lead to an overall speedup of 5.36 on the 8-processor CMP (73% improvement over flat transactions). Closed nesting does not eliminate all overheads as the number of conflicts remains constant and some of them still roll back all the way to the beginning of the outermost transaction. In terms of coding difficulty, closed-nested transactions require additional effort from the programmer in terms of defining the nested transaction boundaries. Nevertheless, the code changes do not affect the correctness of the code. With respect to the other threads, the atomicity boundaries are defined by the outermost transaction. Closed-nested transactions are merely a performance optimization. Hence, a programmer can define their boundaries after identifying the common conflict patterns with a profiling tool like TAPE [2].

Nested transactions typically introduce a small runtime overhead as a few additional instructions must be executed on their boundaries [6]. In this case, we did not observe any slowdown (additional useful time), as the nested transactions are large enough to amortize these overheads. Actually, we observed a small reduction in useful time due to better caching behavior when transactions do not roll back all the way to the beginning of the outermost transaction.

4.3 Open Nesting

Figure 6 shows the transaction-based version of SPECjbb2000 using an open-nested transaction. Specifically, we used an open-nested transaction for the update to the `newID` (lines from 8 to 10) to eliminate the common conflict between all threads operating on a new order. Without the open-nested transaction, all new orders are serialized due to the read-modify-write on the shared `newID` variable. The open-nested transaction allows new order tasks to overlap significantly. When

```
1  TransactionManager::go() {
2      while (workToDo) {
3          switch (transactionType) {
4              case new_order:
5                  atomic {    // begin outermost transaction
6                      // 1. initialize a new order transaction
7
8                      open_atomic {   // begin open-nested transaction
9                          // 2. create a unique order ID and a new order with the ID
10                     }               // commit open-nested transaction
11
12                     // 3. retrieve items and stocks from warehouse
13                     // 4. calculate cost and update warehouse
14                     // 5. record the order for delivery
15                     // 6. print the result of the process
16                 }           // commit outermost transaction
17                 break;
18             ...
19         }
20     }
21 }
```

Figure 6: The SPECjbb200 parallel pseudocode with open-nested transactions..

the open-nested transaction reaches its end (line 10), we do not just merge its write-set (the `newID` counter) to the parent's write-set. We actually commit it to shared memory so that other threads can use the new value of the counter[2]. Hence, two threads executing new order tasks generate a conflict only if their updates to `newID` overlap. This case is rare as the read-modify-write operation is fast and has a low penalty (it causes a rollback to line 8, not line 5). Note that the two threads may later detect a conflict on another object (e.g., `orderTable`), which will cause one of the two to roll back its transaction. When that transaction re-executes, it will get a new `newID`. For SPECjbb2000 this is safe, as order IDs need to be unique but not necessarily sequential. In other cases, an open-nested transaction must be accompanied by compensation code to run if the parent transaction aborts after the nested transaction commits (see [6] for implementation details).

The third bar in Figure 3 shows that adding one open-nested transaction in the original code from Section 4.1 leads to speedup of 4.96q (60% improvement over flat transactions). Even though there are still other conflicts that occur, the open-nested transaction virtually eliminates one of the most frequent sources of conflicts. The disadvantage of open-nested transactions is that the programmer must be very careful when using them as they change the atomicity behavior with respect to other threads. Much better understanding of the code is necessary to decide when an open-nested transaction can be safely used and if compensation code is necessary. For the case of the `newID` counter in SPECjbb2000, using open nesting is relatively simple, but this case does not necessarily generalize. On the other hand, expert programmers can use open nesting to create fast libraries of commonly used objects. In this case, the library would contain an object that returns unique, but not sequential, IDs. Regular programmers can use open nesting through such libraries by understanding the service provided without being exposed to the coding complications of open nesting.

We should note that we could have also eliminated the conflict on `newID` by providing each thread private counters with a sufficient offset so that they do not overlap. Conceptually, this is as difficult as using open nesting for this specific case, as the programmer must recognize that private counters are sufficient and implement an offset scheme that is safe. Moreover, the safety of the offset scheme is dependent on the number of threads used and the length of a SPECjbb2000 run. Hence, in some cases, open nesting can be a viable alternative to privatization. Another interesting alternative is to use a separate flat transaction to create a new ID before entering the main transaction by moving up line 9 over the main transaction. Programmers need to be cautious when splitting a single transaction into smaller ones as they may introduce races due to atomicity breach. In this case, the programmer must notice that there is no dependency between the new order transaction

[2]For more details on the semantics and implementation of open nesting refer to [6]

initialization code (line 6) and new ID creation (line 9).

5 Conclusions

In this paper, we used transactional memory to parallelize the SPECjbb2000 code that operates on a single warehouse. We demonstrated that flat, coarse-grain transactions, which are trivial to use, lead to significant speedups over the sequential version. The frequency and penalty of conflicts between concurrent transactions can be reduced significantly by using nested transactions. Closed-nested transactions are easy to use as they only involve performance tuning and do not affect correctness. Open-nested transactions are more difficult to use as the programmer must ensure that they do not break the atomicity requirements of the application and that compensation code is provided if needed.

There are several more versions of the SPECjbb2000 code that one can construct using different boundaries or combinations for closed and nested transactions. In particular, combining the closed and open nested transactions presented in Sections 4.2 and 4.3 respectively should provide for further performance improvements. However, we believe that the presented versions sufficiently demonstrate the ease-of-use and performance potential of transactional memory with irregular code. They also raise interesting issues with respect to nested transactions in user-level programming.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: A Transactional Application Profiling Environment. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 199–208. June 2005.

[3] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[5] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *19th International Symposium on Distributed Computing*, September 2005.

[6] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006.

[7] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 63–74, Washington, DC, USA, September 2005. IEEE Computer Society.

[8] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[9] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. `http://www.spec.org/jbb2000/`, 2000.