Implementing Atomic Section by Using Hybrid Concurrent Control*

Lei Zhao, Yu Zhang
Department of Computer Science & Technology
University of Science & Technology of China, Hefei, 230027, China
arre@mail.ustc.edu.cn, yuzhang@ustc.edu.cn

Abstract

Atomic section is an important language feather in multi-thread synchronizing. So far, it can only be implemented by using pessimistic or optimistic concurrent control singly. This paper introduces a flexible hybrid concurrent control system which could harmonize the two modes of concurrent control. Accordingly, a new atomic section is proposed as language level support to open an interface for both manual and compiler-assisted optimization.

1. Introduction

Atomic section is an important language feature in multi-thread synchronizing, which guarantees a set of statements could be executed atomically and frees programmer from managing concurrent control strategy. Commonly, atomic sections are implemented by using two distinct modes of concurrent controls: the lock-based pessimistic [4] [8] and the STM(software transaction memory)-based optimistic [1] [6]. It is well known that locking usually lead to coarse concurrent granularity especially for the large and complex data structures with irregular shapes, because fine granularity concurrent algorithms are not only hard to design but liable to cause mistakes, such as deadlock and priority reverse. STM completely eliminates these drawbacks and possesses very fine granularity, however, it inevitably has shortcoming such as high cost of memory space and computing resource, and unable to support irreversible operations, such as I/O.

Traditionally, pessimistic and optimistic concurrent controls are incompatible and unable to work simultaneously within one system. However, considering the balance of scalability and overhead, either of them singly cannot achieve high performance at any time. A more flexible strategy is needed in implementing atomic sections.

*This work is funded in part by gifts from Intel Corporation and National Natural Science Foundation of China under Grant No. 60673126.

This paper introduces a hybrid concurrent control system which can harmonize pessimistic and optimistic modes, letting them cooperate correctly and effectively. Accordingly, a new structure of atomic section is proposed for opening an interface for both manual and compiler-assisted optimization.

The rest of this paper is organized as follows. Section 2 describes the syntax and semantics of the atomic section. Section 3 illustrates the essentials of hybrid concurrent control: conflict definition and avoidance, and operation protocol. Section 4 explains how to handle nesting. Experimental results and interpretation are presented in section 5. Finally we conclude our work in section 6.

2. Atomic Section

```
atom (guard_object_list
[,concurrent_control_mode])
{stmts;}
```

Figure 1. Structure of atomic section.

Figure 1 presents the structure of atomic section. The new keyword *atom* is used to identify a synchronized block, i.e. {*stmts*}, in which another atomic section could be nested.

guard_object_list (shorted as *gol*) declares a list of shared objects. Each of them is called a guard-object. Atomic sections with at least one common guard-object show isolation and atomicity to each other.

concurrent_control_mode is an optional field, which declares the mode of concurrent control to be applied on this atomic section. It can be set to PESS or OPT. PESS means pessimistic concurrent control (P mode), while OPT means optimistic concurrent control (O mode).

Limitation. Most of efforts on determining the concurrent control mode are supposed to be performed automatically by compiler and/or concurrent control system using self-adaptive algorithms, both of which would be our future work. Currently, our experiments in section 5 remain relaying on specifying manually.

3. Hybrid Concurrent Control

The hybrid concurrent control system harmonies pessimistic and optimistic modes, letting them co-work correctively and effectively. In this section, we describe the details in designing hybrid concurrent control system.

3.1. Transaction

Informally, a transaction is one pass of execution of certain atomic section. Following the terminology of Herlihy, Wing [2], and Scott[7], a *transaction* is defined as a sequence of *operations* performed by a single thread. If a transaction adopts pessimistic or optimistic concurrent control, we say it is with P mode or O mode respectively. In our system, operations include:

For transaction with P mode:

- $\mathbf{start^P}$(*gol*): starts the transaction of the related atomic section with *gol*;
- $\mathbf{read^P}$(*so*): reads current value of the shared object *so* from shared memory;
- $\mathbf{write^P}$(*so*,*v*): updates *so* with value *v* by directly writing *v* into shared memory;
- $\mathbf{end^P}$(*gol*): terminates the transaction of the related atomic section with *gol*.

For transaction with O mode:

- $\mathbf{start^O}$(*gol*): starts the transaction of the related atomic section with *gol*;
- $\mathbf{read^O}$(*so*): reads current value of *so* from thread-local memory; if it contains no value of *so*, reading from shared memory;
- $\mathbf{write^O}$(*so*,*v*): updates *so* with the value *v* by writing *v* into thread-local memory;
- $\mathbf{commit^O}$(*gol*,so_1,so_2,...,so_n): terminates the transaction by exiting corresponding atomic section with guard-object list *gol* after updating shared memory using local values of so_i $(1 \leq i \leq n, 0 \leq n)$;
- $\mathbf{abort^O}$(*gol*): terminates the transaction of the related atomic section with *gol* without updating shared memory.

```
Trans →
    (start^P((read^P|write^P)*NestTrans)*end^P) |
    (start^O((read^O|write^O)*NestTrans)*
        (commit^O|abort^O))
NestTrans → Trans|ε
```

Figure 2. Definition of transaction.

Figure 2 describes the form of operations sequence in a transaction. A thread first starts a transaction, performing several read/write operations, and then terminates the transaction. The termination may succeed or fail in O mode but definitely succeed in P mode. NESTTRANS represents the nested part of that transaction. Transactions in different threads inherit the identification of that thread. We use $\mathbf{OP^M_T}$ to represent an operation $\mathbf{OP}$ being performed in transaction T with concurrent control mode M in following part of this paper.

3.2. Conflict

For a program, one pass of execution is represented by a *history*, which is a chronological sequence of operations that performed by all threads according to global system time. A history $\mathcal{H}$ induces a partial order relation $\leq_{\mathcal{H}}$ on operations: for any two operations $\mathbf{op_1}$ and $\mathbf{op_2}$, if $\mathbf{op_1} \leq_{\mathcal{H}} \mathbf{op_2}$, then $\mathbf{op_1}$ precedes $\mathbf{op_2}$. To define conflict, we slightly modify the definition of consistency in [7]: we say a history $\mathcal{H}$ is consistent, if there is operation $\mathbf{read^P_T}(so)$ or $\mathbf{read^O_T}(so)$(T holds gol_1 as its guard-object list), the value of so has not been modified until $\mathbf{end^P_T}(gol_1)$ or $\mathbf{commit^O_T}(gol_1,\ldots)$ appears respectively by transaction S with gol_2 which shares at least one guard-object with gol_1. Any sequence of operations that causes inconsistency raises a conflict. We define conflict by dividing it into three categories: (In the following definitions, for any two transactions T with gol_1 and S with gol_2, there are T $\neq$ S and $gol_1 \cap gol_2 \neq \Phi$.)

Conflict Definition

P vs. P Conflict: In history $\mathcal{H}$, it will raise a P vs. P conflict, if there exists an operation sequence of $\mathbf{read^P_T}(so) \leq_{\mathcal{H}} \mathbf{write^P_S}(so) \leq_{\mathcal{H}} \mathbf{end^P_T}(gol_1)$.

P vs. O Conflict: In history $\mathcal{H}$, it will raise a P vs. O conflict, if there exists an operation sequence of $\mathbf{read^P_T}(so) \leq_{\mathcal{H}} \mathbf{commit^O_S}(gol_2, so, \ldots) \leq_{\mathcal{H}} \mathbf{end^P_T}(gol_1)$ or $\mathbf{read^O_T}(so) \leq_{\mathcal{H}} \mathbf{write^P_S}(so) \leq_{\mathcal{H}} \mathbf{commit^O_T}(gol_1, \ldots)$.

O vs. O Conflict: In history $\mathcal{H}$, it will raise a O vs. O conflict, if there exists an operation sequence

of $\mathbf{read}^{\mathbf{O}}_{\mathbf{T}}(so) \leq_{\mathcal{H}} \mathbf{commit}^{\mathbf{O}}_{\mathbf{S}}(gol_2, so, \ldots)$
$\leq_{\mathcal{H}} \mathbf{commit}^{\mathbf{O}}_{\mathbf{T}}(gol_1, \ldots)$.

Above conflict definition has several drawbacks: 1) increasing possibility to incur deadlock. Because threads will access different shared objects exclusively in uncertain order, any pairs of reversed lock acquiring orders may cause deadlock; 2) increasing overhead of locking. The overhead will rise proportionately to the number of shared objects that have been read. To avoid the difficulty, our system uses a broadened conflict definition.

Broadened Conflict Definition

P vs. P Conflict: In history $\mathcal{H}$, it will raise a P vs. P conflict, if there exists an operation sequence of $\mathbf{start}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{P}}_{\mathbf{S}}(gol_2) \leq_{\mathcal{H}} \mathbf{end}^{\mathbf{P}}_{\mathbf{T}}(gol_1)$.

P vs. O Conflict: In history $\mathcal{H}$, it will raise a P vs. O conflict, if there exists an operation sequence of $\mathbf{start}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{O}}_{\mathbf{S}}(gol_2) \leq_{\mathcal{H}} \mathbf{end}^{\mathbf{P}}_{\mathbf{T}}(gol_1)$ or $\mathbf{start}^{\mathbf{O}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{P}}_{\mathbf{S}}(gol_2) \leq_{\mathcal{H}} \mathbf{commit}^{\mathbf{O}}_{\mathbf{T}}(gol_1, \ldots)$.

O vs. O Conflict: In history $\mathcal{H}$, it will raise a O vs. O conflict, if there exists an operation sequence of $\mathbf{read}^{\mathbf{O}}_{\mathbf{T}}(so) \leq_{\mathcal{H}} \mathbf{commit}^{\mathbf{O}}_{\mathbf{S}}(gol_2, so, \ldots)$
$\leq_{\mathcal{H}} \mathbf{commit}^{\mathbf{O}}_{\mathbf{T}}(gol_1, \ldots)$.

3.3. Keep Consistency

In order to avoid conflicts, conflict avoidance rules in Table 1 are used. Once concurrent control system has detected a critical sequence, which is such sequence that will immediately raise a conflict, it will force the execution to generate the corresponding solved sequence.

For rules A and B, it would raise conflict if an $\mathbf{end}^{\mathbf{P}}_{\mathbf{T}}$ were appended to the sequence, so rules delay the $\mathbf{start}$ of S until $\mathbf{end}^{\mathbf{P}}_{\mathbf{T}}$ is performed. For rules C and D, if T is threatened by dirty read, it is forced to abort.

Implementing rules A and B is straightforward. Every shared object has been bound with a lock. When transaction S starts, it will find that the locks of gol_1 has been hold by transaction T, therefore having to wait until T releases those locks. Implementing rules C and D has a little difficulty because start and read of T are invisible to S, who would be therefore unable to see the critical sequence. This problem is resolved by using visible reader [5]. Every shared object maintains a visible reader list, which records all transactions having read this object. Consequently, for rule D, S is able to detect read of T at the point of $\mathbf{commit}^{\mathbf{O}}_{\mathbf{S}}$. For rule C, T is added as a visible reader of guard-objects in gol_1 while $\mathbf{start}^{\mathbf{O}}_{\mathbf{T}}(gol_1)$ is performed, then start of T is visible to S.

3.4. Operation Protocol

In this section, we describe the operation protocol of hybrid concurrent control. This protocol is implemented according to the conflict avoidance rules; it guarantees conflict-free histories in all executions of multi-thread programs.

Operation Protocol

- Transaction T with P mode must successfully acquire all locks of its guard-objects before start, aborting all visible readers of the guard-objects, then release these locks when exiting.
- Transaction T with O mode can start only when all locks of its guard-objects are free (by acquiring all then releasing all). After starting successfully, T marks itself as a visible reader of the guard-objects.
- Transaction T with O mode can perform $\mathbf{read}^{\mathbf{O}}_{\mathbf{T}}(so)$ and $\mathbf{write}^{\mathbf{O}}_{\mathbf{T}}(so, v)$ only when it has not been aborted. At the earliest performed $\mathbf{read}^{\mathbf{O}}_{\mathbf{T}}(so)$ or $\mathbf{write}^{\mathbf{O}}_{\mathbf{T}}(so, v)$, T marks itself as a visible reader of so.
- Transaction T with O mode must perform self-validating at least one time (the one at $\mathbf{commit}^{\mathbf{O}}_{\mathbf{T}}$) before it can successfully commit. Once being aborted, T restarts without any shared memory updating.
- Transaction T with O mode remains active at $\mathbf{commit}^{\mathbf{O}}_{\mathbf{T}}$, it acquires all locks of its guard-objects, writing back the local copies to shared memory, aborting all visible readers of the objects it wrote, and then releases the locks.

Deadlock avoidance. In the stage of lock acquiring, two transactions might form a waiting cycle due to the improper acquiring order. To avoid deadlock, a transaction is forced to acquire all locks of its guard-objects at one time; otherwise, it releases the locks having already acquired then restarts the process of acquiring.

3.5. Example

In Figure 3, we list four transactions sharing single guard object so_1, where (1) (2) are with P mode and (3) (4) with O mode. The table below describes how the hybrid concurrent control works when each pair of them run concurrently. When (1) starts before (2), (2) will be blocked and wait for the lock of so_1. The second column is similar: (3) will be blocked too. In column three, (3) starts first, and then (1)

	Critical Sequence	Solved Sequence
A	$\mathbf{start}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{P}}_{\mathbf{S}}(gol_2)$	$\mathbf{start}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{end}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{P}}_{\mathbf{S}}(gol_2)$
B	$\mathbf{start}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{O}}_{\mathbf{S}}(gol_2)$	$\mathbf{start}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{end}^{\mathbf{P}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{O}}_{\mathbf{S}}(gol_2)$
C	$\mathbf{start}^{\mathbf{O}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{P}}_{\mathbf{S}}(gol_2)$	$\mathbf{start}^{\mathbf{O}}_{\mathbf{T}}(gol_1) \leq_{\mathcal{H}} \mathbf{start}^{\mathbf{P}}_{\mathbf{S}}(gol_2) \leq_{\mathcal{H}} \mathbf{abort}^{\mathbf{O}}_{\mathbf{T}}(gol_1)$
D	$\mathbf{read}^{\mathbf{O}}_{\mathbf{T}}(so) \leq_{\mathcal{H}} \mathbf{commit}^{\mathbf{O}}_{\mathbf{S}}(gol_1, so, \ldots)$	$\mathbf{read}^{\mathbf{O}}_{\mathbf{T}}(so) \leq_{\mathcal{H}} \mathbf{commit}^{\mathbf{O}}_{\mathbf{S}}(gol_1, so, \ldots) \leq_{\mathcal{H}} \mathbf{abort}^{\mathbf{O}}_{\mathbf{T}}(gol_2)$

Table 1. Conflict avoidance rules. In all rules, $gol_1 \cap gol_2 \neq \Phi$.

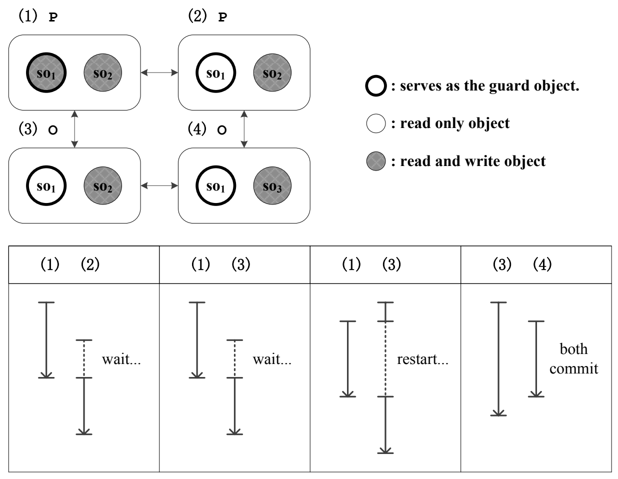

Figure 3. Program behavior under the hybrid concurrent control.

starts. (3) will be aborted and restarts. In sum, transaction with P mode would not interleave with others no matter with P or O mode. However, while (3) and (4) encounter with each other, both of them could commit, because their write sets have no elements in common. This ensures the fine-granularity concurrency of optimistic mode.

4. Nesting

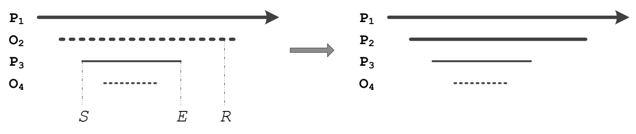

Figure 4. Dealing with mixed nesting.

Nesting is a critical aspect that should be supported in any concurrent control system, in that almost all recursive algorithms and function call may lead to nested transaction.

Let M_i (could be O_i or P_i) be the concurrent control mode of transaction with nested depth i (the outmost is with depth 1). All transactions that nested together could be symbolized by a sequence of $M_1 M_2 \ldots M_i \ldots M_n$ where $1 \leq i \leq n$. All of these sequences are divided into three classes: 1) all of transactions are with O mode, which we denote with $O_1 O_2 \ldots O_i \ldots O_n$; 2) all of transactions are with P mode, which we denote with $P_1 P_2 \ldots P_i \ldots P_n$; 3) mixed nesting, where M_i depends on the mode on depth i.

For case one, we apply closed nesting mode: the inner transaction merely commit its works to the outer one, rather than to shared memory. A drawback of this method is that when an inner transaction is aborted, the whole nested block is requested to be re-executed. However, given it increases the overhead of rollback, it is more practical for the low cost of maintaining information about nesting. For case two, the whole nested blocks are flattened similarly

For case three, a restriction is imposed on the style of mode mixing: for sequence $M_1 M_2 \ldots M_i \ldots M_n$, if M_i is P mode, all M_j that have $1 \leq j \leq i$ must also be P mode, because the sequence may otherwise violate the semantic of abort that once a transaction has been aborted, its effect should be discarded. An example is shown in Figure 4. The left part displays four transactions $P_1 O_2 P_3 O_4$ nested together (we use its mode to represent the transaction for short). Under this case, P_3 starts at the time point S and ends at E, during when it will directly update shared memory. Suppose that if O_2 was then aborted by other thread at R and tried to discard its effect for rollback, it is unable to discard those work done in P_3. To conform to the restriction, when P_3 reaches S, it must rollback to the start of O_2, switching the mode of O_2 to P, then restarts the execution, as showed on the right side in Figure 4.

5. Experiments

Our experiment is carried on simulated shared memory multiprocessor (SMP) environment by using Simics[3]. Configuration of the system is listed bellow:

- The SMP system contains 15 CPUs with Pentium IV instruction set, 20MHz main frequency and split L1 cache: 16K, 4-way, 1-cycle latency. 512MB main memory (100-cycle latency).
- Operating system is Linux Fedora Core 5 (kernel version 2.6.9).
- The JVM is Sun HotSpot JVM 1.6.0 with default options.

The three benchmarks used in the experiments include two variants of integer sets, i.e., a sorted linkedlist and a red-black tree (rbtree for short), and a compounded benchmark combining a linkedlist with an rbtree. Benchmarks support operations: *insert*, *lookup*, and *remove*. *insert* and *remove* will write the shared memory, while *lookup* only reads it.

The number of threads ranges from 1 to 15; every thread operates on rbtree or linkedlist with initial 100 nodes for 10 seconds; the total number of operations is counted as the criteria of performance. The concurrent control system is configured to three types: 1) Pess: configuration "read-P; write-P" (both read and write use P mode); 2) Opt: "read-O; write-O"; 3) Hyb: "read-O; write-P". Write percentage has two levels: 1) heavy write (HW) with %6 read and %94 write; 2) light write (LW) with %94 read and %6 write.

5.1. Performance

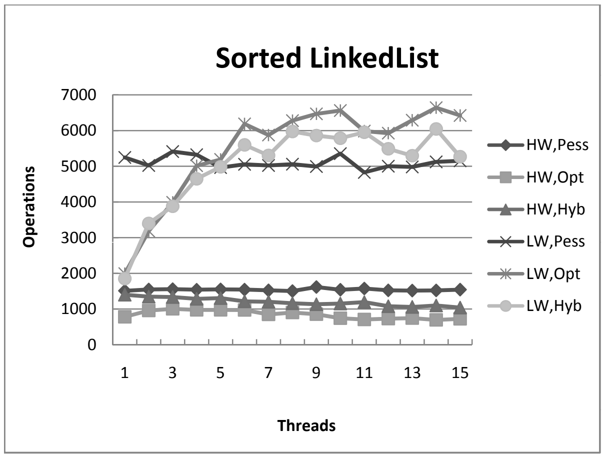

Figure 5. Performance contrast over linkedList.

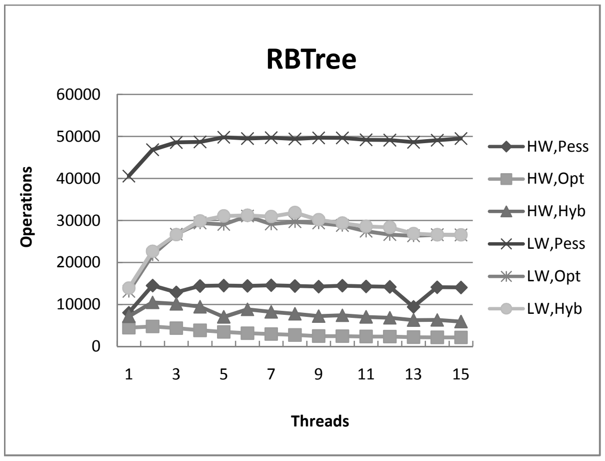

Figure 6. Performance contrast over rbtree.

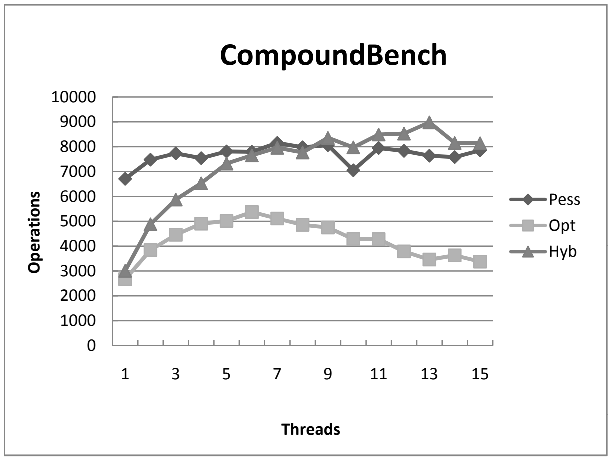

Figure 7. Performance contrast over compoundBench.

Figure 5 and 6 show the performance contrast over rbtree and linkedlist respectively. Pess shows little scalability, but it takes the advantages of incurring lower overhead than Opt does. So when number of threads is small, Pess is much faster than the other two. Because of the intensive data content on HW, Opt cannot benefit from the thread number increasing, thus Pess is always the most efficient configuration on HW.

The write on rbtree is much heavier than on linkedlist due to the balance keeping for the tree, so it limits the potential concurrency of rbtree. The result is that on rbtree, Pess is more effective than Opt not only on HW but on LW as well. However, on LW-linkedlist, Opt is evidently superior to Pess when the number of threads exceeds five. The performance of Hyb is slightly better than Opt on rbtree, however it dominates totally on linkedlist. On HW, its performance approaches equality with that of Opt; on LW, the performance also equates to approximately 80 percent of Pesss. In other words, Hyb gets relative high performance when data contention is intensive meanwhile shows well scalability when data contention is rare.

On compoundbench, every thread performs read or write on linkedlist and rbtree alternately. rbtree is with HW, while linkedlist with LW. The Hyb is configured as "rbtree-P; linkedlist-O". Figure 7 presents the performance and Hyb also shows high scalability. Figure 8 compares the ratio of abort per operation of Opt and Hyb. It illustrates that Hyb could effectively do the same amount of works as Opt could but avoid very high waste of system resource.

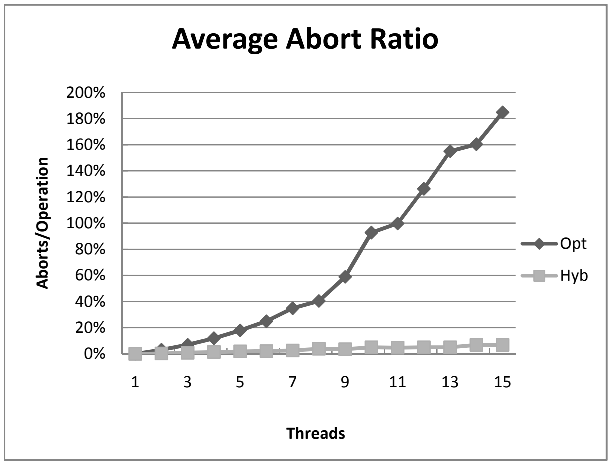

Figure 8. Overhead contrast over compoundedBench.

6. Conclusion

The atomic section described in this paper provides the hybrid concurrent control with language support. As theoretic foundation of the hybrid concurrent control, the conflict definition and solution makes the system possessing low overhead of pessimistic mode and fine scalability of optimistic mode. The experiments show that hybrid concurrent control is very flexible when being applied to different programs. However, so far we just focus on correctly and effectively harmonizing pessimistic and optimistic modes in hybrid concurrent control system and have not developed a method to automatically configure the hybrid mode according to programs. Further, the strategy of conflict avoidance is currently hard coded therefore lacking self-adaptability. It is reasonable to develop various strategies to manage conflicts. These problems are still open and require for further research.

References

[1] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[2] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. In *ACM Transactions on Programming Languages and Systems*, 1990.

[3] P. S. Magnusson, M. Christensson, J. Eskilson, F. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: a full system simulation environment. *IEEE Computer*, pages 50–58, 2002.

[4] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *33rd ACM Symposium on Principles of Programming Languages*, pages 346–358, 2006.

[5] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 1979.

[6] B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[7] M. L. Scott. Sequential specification of transactional memory semantics. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[8] Y. Zhang, J. B. Manzano, and G. R. Gao. Atomic section: concept and implementation. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2005.