第39卷第11期 中国科学技术大学学报 Vol. 39, No. 11
2009年11月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Nov. 2009

Article ID: 0253-2778(2009)11-1224-08

The extension of OpenMP parallel programming model to support transactional memory execution

YANG Xiao-qi[1,2], ZHENG Qi-long[1,2], CHEN Guo-liang[1,2]

(1. *National High Performance Computing Center at Hefei, USTC, Hefei* 230027, *China*;
2. *Department of Computer Science and Technology, USTC, Hefei* 230027, *China*)

Abstract: Although OpenMP is the popular multithread programming model on CMP architecture, OpenMP compilers do not check data dependency, memory access confliction and other problems likely to cause program errors. The traditional lock is applied by programmers to guarantee the correctness of their programs. It is easy to write coarse-grain lock programs, but the parallelism of the program may be lost. On the other hand, potential parallelism of a programs can be found by writing fine-grain lock programs, but it may bring about unwanted problems, such as priority inversion, deadlock, etc. Applying binary instrumentation technology to realize the extension of OpenMP to support transactional memory can effectively alleviale the contradiction between the simplicity and productivity in writing parallel OpenMP programs.

Key words: CMP; OpenMP; transactional memory execution

CLC number: TP311.1; TP338.6 **Document code**: A

扩充 OpenMP 并行编程模型支持事务存储执行

杨晓奇[1,2], 郑启龙[1,2], 陈国良[1,2]

(1. 国家高性能中心, 安徽合肥 230027; 2. 中国科学技术大学计算机科学与技术系, 安徽合肥 230027)

摘要: 虽然 OpenMP 是多核体系结构上的流行多线程并行编程模型, 但是 OpenMP 编译器不检查数据相关性、访问冲突和其他可能导致程序错误执行的问题, 这些问题传统上完全依赖用户使用锁机制来保证程序的正确性. 锁机制的并行编程中存在并行程序效率和并行编程难度的矛盾. 粒度大的锁机制编程容易, 可应用的并行性挖掘比较差; 粒度小的锁机制应用的并行性挖掘较好, 可编程难度大, 容易带来优先权倒置、死锁和锁护航等问题. 通过动态二进制插桩技术, 扩充 OpenMP 支持事务存储执行功能, 可有效缓解 OpenMP 并行编程中并行程序效率和并行编程难度之间矛盾.

关键词: 多核; OpenMP; 事务存储执行

Received: 2008-03-03; **Revised**: 2008-06-02

Foundation item: Supported by the National Natural Science Fund Key Project (60533020), Natural Science Found of Anhui Province (090412068) and Major Project of National Science and Technology (2009ZX01034-001-001-002).

Biography: YANG Xiao-qi, male, born in 1980, PhD candidate. Research field: Parallel and distributed Computing. E-mail: xqyang11@mail.ustc.edu.cn

Corresponding author: CHEN Guo-liang, Academician of the Chinese Academy of Science. E-mail: glchen@ustc.edu.cn

0 Introduction

The room for increase in the processing power of the processor by raising its clock speed is limited by Moore's Law. More recently, there is a growing trend to put more than one processor cores in the same module, instead of creating more complicated single processors. Chip multiprocessor (CMP) is proved to provide comparable or better performance at lower power. The popularity of CMP implementations has a significant impact on the programming model. In order to take advantage of multi-processors' potential computing power, multithreading parallel programming on shard-memory architectures is preferred.

Although OpenMP is the industrial standard for writing multithreading parallel programs on shard-memory architectures, there are no safeguards in either the OpenMP specifications or its implementation to prevent data-dependency from the program code. That is to say, avoiding all the problems such as data access confliction, dead lock, race condition causing wrong execution results should be the responsibility of the programmers. It is more difficult for programmers to use the compiler directive correctly in the complex applications. It is common for programmers to apply conventional lock to guarantee the correctness of their program. Compared with coarse-grain lock programs, fine-grain lock programs may expose more potential parallelism in programs. However it is not easy to write fine-grain lock programs without priority inversion or deadlock problems.

Transactional memory can solve the dilemma between the simplicity and productivity of writing parallel OpenMP programs by abstracting the complexity of concurrency shared data to simplify parallel program development. The extension of OpenMP to support transactional memory execution combines the popular parallel multithread programming model (OpenMP) and intuitive shared data programming model (transactional memory execution). Programmers only need to care about where should primitive instruction be used, instead of thinking about which mechanic should be applied.

1 Related work

OpenMP[1] is an application programming interface (API) which supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines and environment variables that influence run-time behavior. OpenMP offers a set of low-level primitives around locks and the high-level critical construct to protect the access to shared data. However, dealing with complex shared-data access using low-level primitives may cause many unwanted problems, such as dead lock, priority inversion and lock convoying.

Transactional memory (TM)[2] is a crucial mechanism to tackle this problem by abstracting away the complexities associated with concurrent access to shared data where multiple threads need to simultaneously access shared memory locations atomically. TM makes it relatively easy to develop parallel programs. Fig. 1 describes a basic transactional memory model using the checkpoint interface. A transaction must check its program state upon beginning a transaction, in case it needs to abort the transaction and roll back execution. All memory access modified by the transaction should be recorded to detect conflicts between

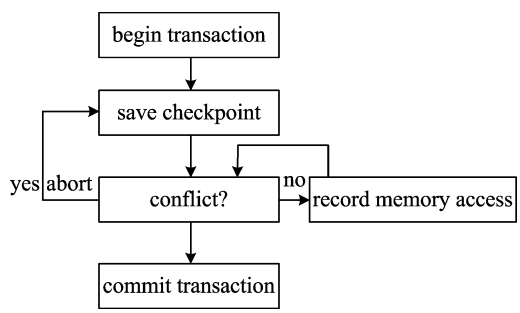

Fig. 1　Transactional memory execution model

concurrent transactions. If a conflict is detected and the thread is chosen to abort, it must use its log to restore all the memory locations it has modified and revert back to its saved checkpoint to retry the transaction. If the thread completes the transaction without detecting any conflicts, it can effectively commit its changes by discarding the checkpoint and log.

Transactions have long been used for fault tolerance in databases and distributed systems[3]. Transactions memory execution[4] provides an intuitive model for reasoning about coordinate access to shared data, which comprises a series of read and write operations that provide the properties of failure atomicity, consistency and durability. Herlihy et al.[5] introduced hardware transactional memory (HTM) and showed that bounded-size atomic transactions that are short enough to be completed without context switching could be supported using simple additions to the cache mechanisms of existing processors. Unbounded HTM[6] aims to overcome the disadvantages of simple bounded HTM designs by allowing transactions to commit even if they exceed on-chip resources and/or run for longer than a thread's scheduling quantum. Herlihy et al[7] have designed a practical software transactional memory, which is obstruction-free and requires only the readily available compare-and-swap (CAS) instruction. Damron et al.[8] apply the HyTM approach to provide an STM implementation that does not depend on hardware support beyond what is widely available today, and also to provide the ability to execute transactions using whatever HTM support is available in such a way that the two types of transactions can coexist correctly.

The fusion of OpenMP with TM can simplify the development of parallel programs for CMP. It makes it possible for programmers to apply popular parallel programming model coupled with a more intuitive way of writing shared-data programs. Programmers only need to care about where should primitive instruction be used, instead of thinking about which mechanic should be applied. Nebelung[9] system brings together TM with OpenMP. It can be easily customized to work with STM, HTM and HyTM systems. However it has to do the modification of compilers. Binary instruction technology is applied in the paper to realize the combination OpenMP and TM, instead of modifying compilers. It provides the inspiration for the new transaction memory hardware design.

2 Design and implementation

As described in Fig. 2, the instruction and analysis routines are instrumented with OpenMP program through binary instrumentation tool Pin. Pin[10] was designed to provide functionality similar to the popular ATOM toolkit. Unlike ATOM, Pin does not instrument an executable statically by rewriting it, but rather adds the code dynamically while the executable is running. This also makes it possible to attach Pin to an already running process. Pin provides a rich API that abstracts away the underlying hardware instructions and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves/restores some registers before/after they are overwritten by the injected codes, so that the application may continue running under its normal context. Limited access to symbol and debug information is available as well.

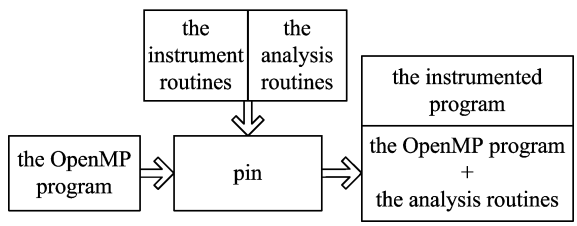

Fig. 2 The system structure

2.1 The transaction execution program structure

Fig. 3 presents the structure of the OpenMP program and the program using transaction memory execution. begin_transaction ( ) registers

a transaction, and all the memory instructions inside the transactions are recorded to detect conflict between concurrent transactions. By examining other registered transactions' record information, a thread can ensure that it is not modifying a memory location accessed by another transaction or accessing a memory location modified by another transaction. Once a conflict is detected, a thread is chosen to be the winner to continue running according to the specific arbitration policy, while others are to be aborted using abort _ transaction ( ). If the thread completes the transaction without detecting any conflicts, it can effectively commit its changes and be unregistered using end_transaction ( ).

```
original program:

#pragma omp parallel
for (i=omp_get_thread_num();
    i<
   N; i + = omp _ get _ num _
   threads())
{
...
   function(i);
   ...
}
```

```
transformed program:

#pragma omp parallel parallel
for (i=omp_get_thread_num();
    i<N; i + = omp _ get _ num _
    threads())
) {
   begin_transaction();
     ...
   function(i);
   if((conflicts) && loser of
   arbration())
   then abort_transaction()
     goto begin_transaction
     ...
   else end_transaction();
}
```

Fig. 3　Transforming OpenMP program in transaction execution program

2.2　The instrument and analysis routines

The main functions of the tool are: supervising the OpenMP program, recording the comprehensive, detailed memory access information, realizing the transaction execution roll back and recovery semantics. It supports lazy/eager conflict checking algorithms, the efficient arbitration algorithms and the log functions. In order to control the OpenMP program, the instrumentation should be done for the API and memory access instruction, as described in Tab. 1.

Tab. 1　The analysis routines

analysis routines	function
before_begin_transaction	save the checkpoint for roll back
before_end_transaction	commit the transaction without conflict
before_abort_transaction	roll back to the save checkpoint

2.3　Memory access conflicts detection

Before and after each memory access, the before_memory_access( ) and after_memory_access ( ) analysis routines are instrumented respectively. For normal instructions, the after_memory_access( ) should be instrumented before its following instruction; For unconditional jump instruction (such as the subroutine call, return instruction), the after_memory_access should be instrumented at the position where it is to be jumped to; For conditional branch instruction, the after_memory_access ( ) should be the success and fall through respectively. There are two memory access conflict detection mechanisms applied here: lazy memory access conflict detection and eager memory access conflict detection.

Algorithm 2.1　Checking confliction algorithm

input:

is_writing: write instruction or no.

addr:: the valid memory write address.

output: none

```
void detect_and_deal_with_conflictions(bool is_writing,
ADDRINT addr)
{
  if (current_thread. is_aborting) {
    abort_current_transaction();
    if (number of transactions > 0) {
        if (number of transactions > 1 or not in
          transaction) {
          detect conflictions with other transactions,
          put the conflicting threads in set conflicts;
            winner = arbitrate _ conflictions ( current _
            thread, conflicts);
          if (winner ! =current_thread)
            abort_current_transaction();
          else /* set aborting flags of conflicting threads */
            for each thread t in conflicts
              t. is_aborting=true;
    }
```

```
  }
}
```

According to the difference of mechanics of detecting conflictions among transactions, there are lazy memory access conflict detection and eager memory access conflict detection. Lazy memory access conflict detection is similar to the hardware mechanics of LogTM[11,12]. The memory access conflict detection is stalled until the transaction is going to be committed. This requires all the memory modification of one transaction be invisible to all others. Therefore after recording the new value of one variable, the old value should be restored. Before committing one transaction, the following equation should be checked. For the transaction i and j, the read set (read_set) and write set (write_set) should satisfy the following equation. Otherwise there are conflictions happen.

$$(\text{read\_set}_i \bigcap \text{write\_set}_j) \bigcup (\text{write\_set}_i \bigcap \text{read\_set}_j)(\text{write\_set}_i \bigcup\bigcap \text{write\_set}_j) = \emptyset \quad (1)$$

Algorithm 2.2 Lazy recording memory access algorithm

input: is_reading: read instruction or not.
raddr: the address of read instruction.
is_writing: write instruction or not.
waddr: the address of write instruction.

output: read_set: the set of the current read instruction.
write_set: the set of the current write instruction.

```
void before_ memory_access(bool is_reading, ADDRINT
                           raddr,
                           bool is_writing, ADDRINT
                           waddr)
{
  lock(memory_access_mutex);
  if (is_reading && is_writing && raddr==waddr) {
    if (has previous write in current transaction) {
    raddr=write_set[raddr]. new_value
      /* previous write value */;
    } else {
    write_set[raddr]. old_value= *raddr;
    }
    read_set[raddr]= *raddr;
  } else if (is_reading) {
    if (has previous write in current transaction) {
      *raddr=write_set[raddr]. new_value
      /* previous write value */;
    }
    read_set[raddr]= *raddr;
    } else if (is_writing) {
     write_set[waddr]. old_value = *waddr /* old
value */;
    }
  }
}
void after_memory_access(bool is_reading, ADDRINT
                                raddr,
                                bool is_writing, ADDRINT
                                waddr)
{
  if ( is_reading && is_writing && raddr==waddr) {
     write_set[raddr]. new_value = *raddr /* new
value */;
    *raddr=write_set[raddr]. old_value;
  } else if (is_reading) {
    if (has previous write in current transaction)
    *raddr=write_set[raddr]. old_value;
    } else if (is_writing) {
     write_set[waddr]. new_value = *waddr /* new
value */;
    *waddr=write_set[waddr]. old_value;
    }
  unlock(memory_access_mutex);
}
```

Eager memory access conflict detection is similar to the hardware mechanics of TCC[13]. Before executing each memory access instruction of each transaction, the following equation should be checked. If the equation is not satisfied, conflictions happen.

$$\text{addr} \notin \text{trans}_j.\text{write\_set and (not is\_writing or addr} \notin \text{trans}_j.\text{read\_set}) \quad (2)$$

Algorithm 2.3 Eager recording memory access algorithm

input: is_reading: read instruction or not.
raddr: the address of read instruction.
is_writing: write instruction or not.

waddr: the address of write instruction.

output: read_set: the set of the current read instruction.

write_set: the set of the current write instruction.

```
void before _ spec _ memory _ access ( bool is _ reading,
                                       ADDRINT raddr,
                                       bool is _ writing,
                                       ADDRINT waddr)
{
  detect & deal with memory access conflictions with other
  threads;
  if (is_reading)
    read_set[raddr]= * addr;
  else / * is_writing * /
    if (is first write in current transaction)
      write_set[waddr]. old_value= * waddr;
}
```

2.4 Arbitration policies

Arbitration policies are applied to select winner traction among conflicting transaction sets. The random policy selects the winner from the conflicting transactions randomly. The timestamp policy checks these transactions' timestamp and regards the winner as the youngest transaction when it encounters an opposing transaction. The workload policy keeps traces of how much work a transaction has done in terms of the number of write instructions that a transaction has executed.

Algorithm 2.4 Arbitrating conflictions algorithm

input: current_thread: the current thread.

thread-set conflicts: the conflicting threads set.

output: the winner transaction.

```
thread arbitrate_conflictions (thread current_thread, thread-
set conflicts, )
{
  if ( select_criterial=random)
  {
    return winner=random(conflicts);
  }
  else if ( select_criterial==timestamp)
  {
    return winner={thread | thread→timestamp=min (
    thread→timestamp | thread (-conflicts)
  }
  else if ( select_criterial==workload)
  {
    return winner={thread | thread→workload=max (
    thread→workload | thread (-conflicts)
  }
}
```

2.5 Transaction commission and abortion

end _ atomic ( ) is applied to commit a transaction. If the transaction does not conflict with other transactions, then what it has modified is confirmed. Transaction abort happens when there is a conflict between transactions. The loser of the arbitration aborts the current transaction. Transaction abortion is realized by two steps. The first step is to recover what the transaction has modified, and the second step is to roll back to the beginning of the transaction for the next try. pin_SaveContext, pin_ExecuteAt, pin_SaveCheckpoint and pin _ resume are applied to realize the mechanics.

```
#define begin_transaction( ) save_check_point();
#define abort_transaction( )\
  current_transaction. restore_memory( );
  current_thread. is_aborting=true;
  resume_execution(); //resume after save_check_point
```

2.6 Transaction execution log

The execution information is important for guaranteeing the correction of programs. In addition, the log information is also necessary for the evaluation of conflict checks and arbitration algorithms. The log information includes the run time, the total number of conflicts, the race condition variables, etc.

3 Evaluation

The case of transactional memory execution of OpenMP is as follows:

```
#define N 300000
int primes[N];
int main(int argn, char * * argv)
{
```

```
int i,total=0;
#pragma omp parallel parallel
for (i=omp_get_thread_num(); i<N; i+=omp_get_
    num_threads())
{
  begin_transaction();
  if ( is_prime(i) ) {
    primes[total]=i;
    total=total+1;//
  }
  end_transaction();
}
```

There are no safeguards in either the OpenMP specifications or its implementation to avoid race conditions from the program code. Many programmers write parallel applications that have race conditions during the development cycle, either consciously or unconsciously. A data race condition[14] occurs when two or more threads in a single process access the same memory location concurrently, of which at least one of the accesses is for writing, and the threads are not using any exclusive locks to control their accesses to that memory. When these three conditions hold, the order of accesses is non-deterministic. Therefore each run can give different results depending on the order of the accesses.

In order to verify that the tool can detect and avoid race condition in OpenMP, the common example with race condition from the tutorial of using Sun data race detection tool[15] is selected. Through checking the memory access record information of transactions, the race condition variable "total" is found due to its being accessed by more than one thread at run time. As described in the case above, programmers only need to do a little modification of the source program to prevent race condition from occurring. Therefore, the extension of OpenMP to support transactional memory can solve the dilemma between the simplicity and productivity of writing parallel OpenMP programs.

Based on the correct program results and the eager memory access conflict detection mechanisms, arbitration algorithms are compared according to the variety number of threads. As can be seen in Fig. 4, work-related arbitration policy is the best because the rolling back threads have to clean up what they have modified before.

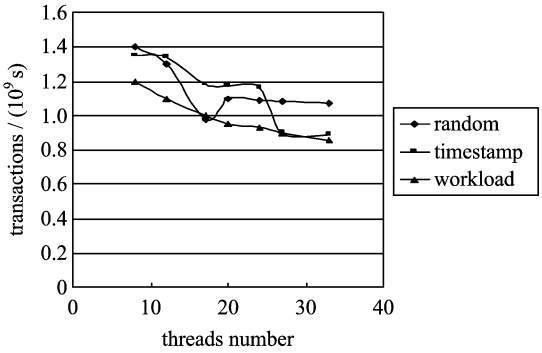

Fig. 4 The comparison of different arbitration policies

4 Conclusion

The extension of OpenMP to support transactional memory can solve the dilemma between simplicity and productivity of writing parallel OpenMP programs. Different from the previous technologies combining transactional memory and OpenMP, the tool presented in the paper is based on the dynamic binary instrumentation technology. Therefore there is no need to recompile or re-link the source code, and the code is discovered at runtime. In addition, transactional execution is applied to reduce the common problems that lock is prone to. These include but are limited to deadlock, lock convoying and priority inversion.

The slowdown of the tool is caused by the instrumentation for each individual memory access instructions and tremendous conflict checks. The software is conductive to the design of future hardware transaction memory, that will reduce slowdown significantly.

References

[1] OpenMP Architecture Review Board. OpenMP Application Program Interface [M]. 2005.

[2] Larus J, Rajwar R. Transactional Memory [M]. CA: Morgan Claypool, 2006.

[3] Gray J, Reuter A. Transaction Processing: Concepts and Techniques [M]. CA: Morgan Kaufmann Publishers, 1992.

[4] Rajwar R, Goodman J. Transactional execution: toward reliable, high performance multithreading [J]. IEEE Micro, 2003, 23(6): 117-125.

[5] Herlihy M, Moss J E B. Transactional memory: architectural support for lock-free data structures[C]// Proceedings of the 20th Annual International Symposium on Computer Architecture. New York: ACM Press, 1993: 289-300.

[6] Israeli A, Rappoport L. Disjoint-access parallel implementations of strong shared memory primitives [C]// Proceedings of the13th annual ACM symposium on Principles of Distributed Computing. Los Angeles: ACM Press, 1994: 151-160.

[7] Harris H, Fraser H. Language support for lightweight transactions [C]// Proceedings of 18th annual SIGPLAN conference on Object-oriented Programming, Systems, Languages, and applications. New York: ACM Press, 2003: 388-402.

[8] Damron P, Fedorova A, Lev Y et al. Hybrid transactional memory [C]// Proceedings of 12th International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose: ACM Press, 2006: 336-346.

[9] Milovanović M, Ferrer R, Gajinov V, et al. Multithreaded software transactional memory and OpenMP[C]// Proceedings of Memory performance: Dealing with Applications, Systems and Architecture. Brasov, Romania: ACM Press, 2007: 81-88.

[10] Luk C, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. SIGPLAN Notices, 2005, 40(6): 190-200.

[11] Moore K E, Hill M D, Wood D A. Thread-level transactional memory[R]. Technical Report: CS-TR-2005-1524, Department of Computer Sciences, University of Wisconsin, 2005.

[12] Moore K E, Bobba J, Moravan M J, et al. LogTM: Log-based transactional memory [C]// Proceedings of 12th Annual International Symposium on High Performance Computer Architecture. 2006: 254-265.

[13] Hammond L, Wong V, Chen M, et al. Transactional memory coherence and consistency [J]. ACM SIGARCH Computer Architecture News, 2004,32(2): 102-113.

[14] Bishop M. Race conditions, files, and security flaws; or the tortoise and the hare redux [R]. Technical Report CSE-95-9, Department of Computer Science, University of California at Davis, 1995.

[15] Lint L. Static data race and deadlock detection tool for C [EB/OL]. http://developers. sun. com/solaris/articles/locklint. html.