6.895 Theory of Parallel Systems 10-08-03

Project Proposal

Jim Sukha

1 Summary

We propose to provide modify the Cilk compiler and construct an run-time system to support atomic transactions in Cilk. We hope to use this system to investigate the behavior of different implementations of transactional memory and identify potential directions for further research.[1]

2 Introduction

This proposed project deals with atomic transactions and transactional memory.

2.1 Atomic Transactions

Once again, we begin by considering the following Cilk program.

```
int x;
cilk void foo(void) {
     x = x + 1;
     return;
}
cilk int main(void) {
     x = 0;
     spawn foo();
     spawn foo();
     sync;
     printf("x is %d\n", x);
     return 1;
}
```

As we discussed earlier in the class, there is a determinacy race because the two spawned threads both try to modify `x`. We would like the body of `foo` to execute atomically, so that one execution of `foo` always completes before the other begins. It would be convenient if a programmer could simply alter the definition of `foo` to reflect this.

```
cilk void foo(void) {
   xbegin
       x = x + 1;
   xend
}
```

[1]I am collaborating with Tushara Karunaratna on this project, but he is not taking this class.

In this code, `xbegin` and `xend` are Cilk keywords that signify the beginning and the end of a transaction. Everything between these two keywords constitutes a section of code that is executed atomically. Our goal is to modify the Cilk language to provide this functionality.

2.2 Transactional Memory

The traditional method for achieving atomicity in the previous example is to use obtain the appropriate locks before beginning a transaction and releasing them when the transaction is complete. In programs where shared memory conflicts are unlikely to occur, using locks may generate unnecessary overhead. The program may spend a lot of time locking and unlocking a value which is almost always used by only one processor anyway. One lock-free alternative is to use transactional memory. In this system, the default option is to execute each transaction. If, while executing, a transaction detects a conflict with another transaction, then one or both of the transactions aborts. Otherwise, if there are no conflicts, at the end of the transaction, all changes that were made are committed to memory.

For this project, we plan to implement a runtime system that simulates transactional memory. To do this, we must identify all load and store operation in the user code, and modify them so that we can support an abort or commit operation. For the previous example, this requires a code transformation:

```
cilk void foo(void) {
   AtomicContext ac = createAtomicContext();
   addMemoryLocation(ac, &x, sizeof(x));

   label attempt_transaction:
      initTransaction();

      ({ int temp_x;
         if (!LD(temp_x, x, ac)) goto failed;
         temp_x = temp_x + 1;
         if (!ST(x, temp_x, ac)) goto failed;
         temp_x;
        });
      goto commit;

   label failed:
      doAbort(ac);
      doBackoff(ac);
      goto attempt_transaction;

   label commit:
      doCommit(ac);
      destroyAtomicContext(ac);
}
```

The original code, `x = x + 1`, has been modified so that the load and store operations are done through functions that will be provided by our runtime system. The rest of the added code handles the beginning and end of the transaction.

3 Project Outline

There are three main components to the project, each with a number of issues that must be addressed:

1. *Compiler Support*: The Cilk compiler must be modified to support the `xbegin` and `xend` keywords. This involves modifying the parser and Cilk grammar to accept the new language constructs. The compiler must also do the code transformations necessary to support transactions. This involves manipulation of the abstract syntax tree generated in `cilk2c`, the program that translates Cilk to C. We have both worked with the `cilk2c` before, so hopefully this step will not be too time-consuming.

2. *Runtime Support*: The runtime system must provide space for transactions to backup the memory locations they modify, so that the original values can be restored in case an abort occurs. The system must also provide a mechanism for detecting when two transactions are in conflict. For the initial implementation, we plan to use a hash table to store a lock and the backup value for each memory location accessed during a transaction. For aborted transactions, we can implement a simple backoff algorithm. As time permits, we hope to try more complicated schemes for the runtime system.

3. *Testing and Applications*: To test the correctness and validity of the system, we can use simple Cilk programs with competing transactions. However, hopefully, we can also find a non-contrived program that might benefit from transactional memory.

After the modifications to the compiler are done, the project itself is somewhat open-ended. The project seems flexible enough to fit within the scope of the course. The amount of experimentation we can do in the later stages can be modified depending on the time required to alter the compiler and implement the initial runtime system.

References

[1] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture*, 1993.

[2] N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994.