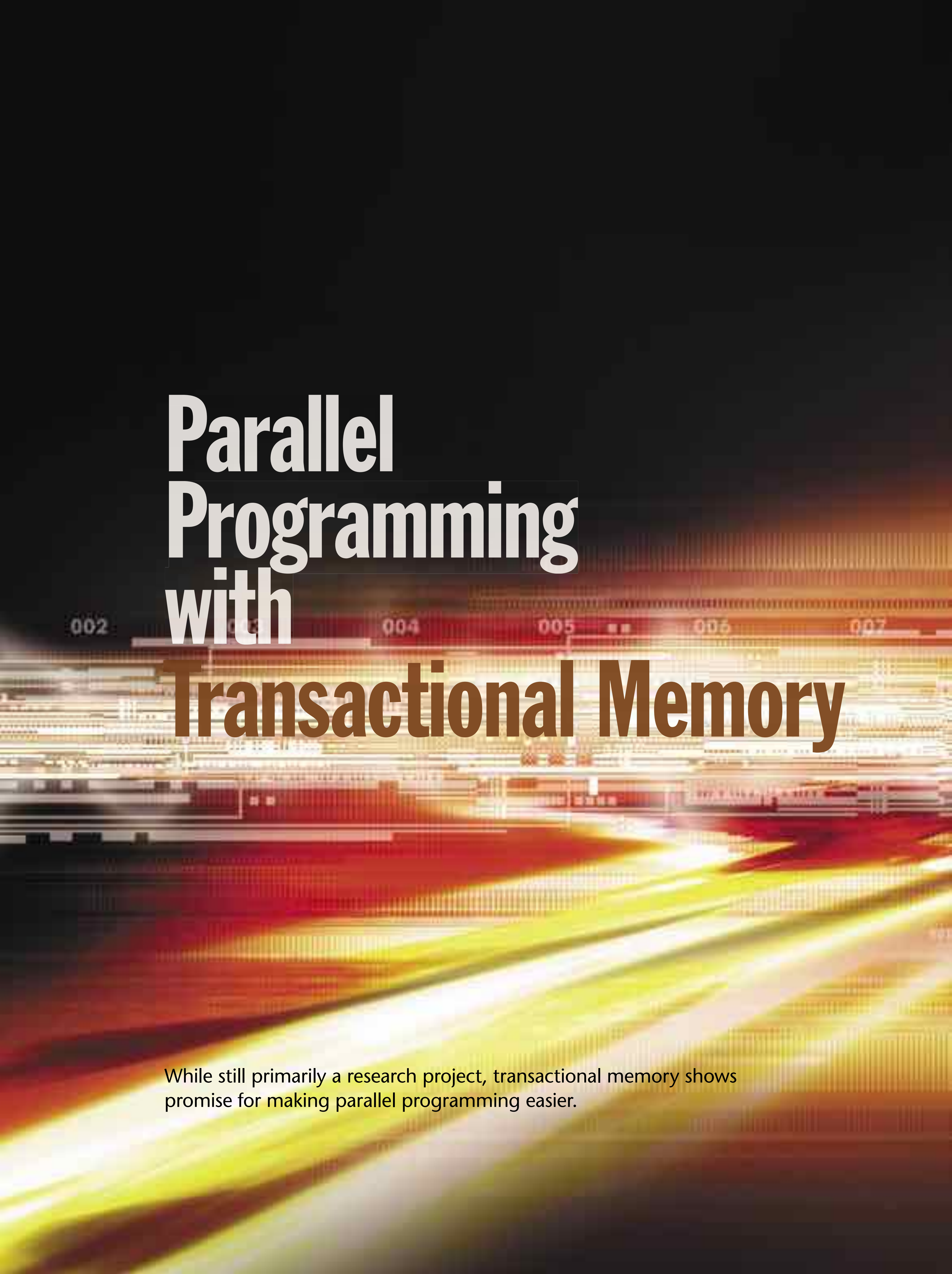
Parallel Programming with Transactional Memory



While still primarily a research project, transactional memory shows promise for making parallel programming easier.

Ulrich Drepper, Red Hat

With the speed of individual cores no longer increasing at the rate we came to love over the past decades, programmers have to look for other ways to increase the speed of our ever-more-complicated applications. The functionality provided by the CPU manufacturers is an increased number of execution units, or CPU cores.

To use these extra cores, programs must be parallelized. Multiple paths of execution have to work together to complete the tasks the program has to perform, and as much of that work as possible has to happen concurrently. Only then is

it possible to speed up the program (i.e., reduce the total runtime). Amdahl's law expresses this as:

$$\frac{1}{(1-P) + P/S}$$

Here P is the fraction of the program that can be parallelized, and S is the number of execution units.

SYNCHRONIZATION PROBLEMS

This is the theory. Making it a reality is another issue. Simply writing a normal program by itself is a problem, as can be seen in the relentless stream of bug fixes available for programs. Trying to split a program into multiple pieces that can be executed in parallel adds a whole dimension of additional problems:

- Unless the program consists of multiple independent pieces from the onset and should have been written as separate programs in the first place, the individual pieces have to collaborate. This usually takes the form of sharing data in memory or on secondary storage.
- Write access to shared data cannot happen in an uncontrolled fashion. Allowing a program to see an inconsistent, and hence unexpected, state must be avoided at all times. This is a problem if the state is represented by the content of multiple memory locations. Processors are not able to modify an arbitrary number (in most cases not even two) of independent memory locations atomically.
- To deal with multiple memory locations, "traditional" parallel programming has had to resort to synchronization. With the help of mutex (mutual exclusion) directives, a program can ensure that it is alone in executing an operation protected by the mutex object. If all read or write accesses to the protected state are performed while holding the mutex lock, it is guaranteed that the program will never see an inconsistent state. Today's programming environments (e.g., POSIX) allow for an arbitrary number of mutexes to coexist, and there are special types of mutexes that allow for multiple readers to gain access concurrently. The latter is allowed since read accesses do not change the state. These mechanisms allow for reasonable scalability if used correctly.
- Locking mutexes open a whole new can of worms, though. Using a single program-wide mutex would in most cases dramatically hurt program performance by decreasing the portion of the program that can run in parallel (P in the formula). Using more mutexes increases not only P, but also the overhead associated with locking and unlocking the mutexes. This is especially problematic if, as it should be, the critical regions are only lightly contended. Dealing with multiple mutexes also means the potential for deadlocks exists. Deadlocks happen if overlapping mutexes are locked by multiple threads in a different order. This is a mistake that happens all too easily. Often the use of mutexes is hidden in library functions and not immediately visible, complicating the whole issue.

THE PROGRAMMER'S DILEMMA

The programmer is caught between two problems:

- Increasing the part of the program that can be executed in parallel (P).
- Increasing the complexity of the program code and therefore the potential for problems.

An incorrectly functioning program can run as fast as you can make it run, but it will still be useless. Therefore, the parallelization must go only so far as not to introduce problems of the second kind. How much parallelism this is depends on the experience and knowledge of the programmer. Over the years many projects have been developed that try to automatically catch problems related to locking. None is succeeding in solving the problem for programs of sizes that appear in the real world. Static analysis is costly and complex. Dynamic analysis has to depend on heuristics and on the quality of the test cases.

For complex projects it is not possible to convert the whole project at once to allow for more parallelism. Instead, programmers iteratively add ever more fine-grained locking. This can be a long process, and if the testing of the intermediate steps isn't thorough enough, problems that are not caused by the most recently added set of changes might pop up. Also, as experience has shown, it is sometimes very hard to get rid of the big locks. For an example, look at the BKL (big kernel lock) discussions on the Linux kernel mailing list. The BKL was introduced when Linux first gained SMP (symmetric multiprocessing) support in the mid-90s, and we still haven't gotten rid of it in 2008.

More and more people have come to the conclusion that locking is the wrong approach to solving the consistency issue. This is especially true for programmers who are not intimately familiar with all the problems of parallel programming (which means almost everybody).

LOOKING ELSEWHERE

The problem of consistency is nothing new in the world of computers. In fact, it has been central to the entire solution in one particular area: databases. A database, consisting of many tables with associated indexes, has to be updated atomically for the reason already stated: consistency of the data. It must not happen that one part of the update is performed while the rest is not. It also must not happen that two updates are interleaved so that in the end only parts of each modification are visible.

The solution in the database world is transactions. Database programmers explicitly declare which database operations belong to a transaction. The operations performed in the transaction can be done in an arbitrary order and do not actually take effect until the transaction is committed. If there are conflicts in the transaction (i.e., other operations are concurrently modifying the same data sets), the transaction is rolled back and has to be restarted.

The concept of the transaction is something that falls out of most programming tasks quite naturally. If all changes that are made as part of a transaction are made available atomically all at once, the order in which the changes are added to the transaction does not matter. The lack of a requirement to perform the operations in a particular order helps tremendously. All that is needed is to remember to modify the data sets always as part of a transaction and not in a quick-and-dirty, direct way.

TRANSACTIONAL MEMORY

The concept of transactions can be transferred to memory operations performed in programs as well. One could of course regard the in-memory data a program keeps as tables corresponding to those in databases, and then just implement the same functionality. This is rather limiting, though, since it forces programmers to dramatically alter the way they are writing programs, and systems programming cannot live with such restrictions.

Fortunately, this is not needed. The concept of TM (transactional memory) has been defined without this restriction. Maurice Herlihy and J. Eliot B. Moss in their 1993 paper[1] describe a hardware implementation that can be implemented on top of existing cache coherency protocols reasonably easily.[2]

The description in the paper is generic. First, there is no need to require that transactional memory be implemented in hardware, exclusively or even in part. For the purpose mentioned in the paper's title (lock-free data structures), hardware support is likely going to be a must. But this is not true in general, as we will see shortly. Second, the description must be transferred to today's available hardware. This includes implementation details such as the possible reuse of the cache coherency protocol and the granularity of the transactions, which most likely will not be a single word but instead a cache line.

Hardware support for TM will itself be mostly interesting for the implementation of lock-free data structures. To implement, for example, the insert of a new element into a double-linked list without locking, four pointers have to be updated atomically. These pointers are found in three list elements, which means that it is not possible to implement this using simple atomic operations. HTM (hardware TM) provides a means to implement atomic operations operating on more than one memory word. To provide more general support for transactional memory beyond atomic data structures, software support is needed. For example, any hardware implementation will limit the size of a transaction. These limits might be too low for nontrivial programs or they might differ among implementations. Software can and must complete the HTM support to extend the reach of the TM implementation meant to be used for general programming.

Software can and must complete the HTM support to extend the reach of the TM implementation meant to be used for general programming.

This has been taken a step further. Because today's hardware is mostly lacking in HTM support, STM (software TM) is what most research projects are using today. With STM-based solutions it is possible to provide interfaces to TM functionality, which later could be implemented in *hybrid TM* implementations, using hardware if possible. This allows programmers to write programs using the simplifications TM provides even without HTM support in the hardware.

SHOW ME THE PROBLEM

To convince the reader that TM is worth all the trouble, let's look at a little example. This is not meant to reflect realistic code but instead illustrates problems that can happen in real code:

```
long counter1;
long counter2;
time_t timestamp1;
time_t timestamp2;

void f1_1(long *r, time_t *t) {
  *t = timestamp1;
  *r = counter1++;
}

void f2_2(long *r, time_t *t) {
  *t = timestamp2;
  *r = counter2++;
}

void w1_2(long *r, time_t *t) {
  *r = counter1++;
  if (*r & 1)
    *t = timestamp2;
}

void w2_1(long *r, time_t *t) {
  *r = counter2++;
  if (*r & 1)
    *t = timestamp1;
}
```

Assume this code has to be made thread-safe. This means that multiple threads can concurrently execute any of the functions and that doing so must not produce any invalid result. The latter is defined here as return counter and timestamp values that don't belong together.

It is certainly possible to define one single mutex lock and require that this mutex be taken in each of the four functions. Verifying that this would generate the expected results is easy, but the performance is potentially far from optimal.

Assume that most of the time only the functions f1_1 and f2_2 are used. In this case there would never be any conflict between callers of these functions: callers of f1_1 and f2_2 could peacefully coexist. This means that using one single lock slows down the code unnecessarily.

So, then, use two locks. But how to define them? The semantics would have to be in the one case "when counter1 and timestamp1 are used" and "when counter2 and timestamp2 are used," respectively. This might work for f1_1 and f2_2, but it won't work for the other two functions. Here the pairs counter1/timestamp2 and counter2/timestamp1 are used together. So we have to go yet another level down and assign a separate lock to each of the variables.

Assuming we would do this, we could easily be tempted to write something like this (only two functions are mentioned here; the other two are mirror images):

```
void f1_1(long *r, time_t *t) {
  lock(l_timestamp1);
  lock(l_counter1);

  *t = timestamp1;
  *r = counter1++;
}

void w1_2(long *r, time_t *t) {
  lock(l_counter1);

  *r = counter1++;
  if (*r & 1) {
    lock(l_timestamp1);
    *t = timestamp2;
    unlock(l_timestamp1);
  }

  unlock(l_counter1);
}
```

The code for w1_2 in this example is wrong. We cannot delay getting the l_timestamp1 lock because it might produce inconsistent results. Even though it might be slower, we always have to get the lock:

```
void w1_2(long *r, time_t *t) {
  lock(l_counter1);
  lock(l_timestamp1);

  *r = counter1++;
  if (*r & 1) {
    *t = timestamp2;

  unlock(l_timestamp1);
  unlock(l_counter1);
}
```

It's a simple change, but the result is also wrong. Now we try to lock the required locks in w1_2 in a different order from f1_1. This potentially will lead to deadlocks. In this simple example it is easy to see that this is the case, but with just slightly more complicated code it is a very common occurrence.

What this example shows is: (1) it is easy to get into a situation where many separate mutex locks are needed to allow for enough parallelism; and (2) using all the mutex locks correctly is quite complicated by itself.

As can be expected from the theme of this article, TM will be able to help us in this and many other situations.

REWRITTEN USING TM

The previous example could be rewritten using TM. In the following example we are using nonstandard extensions to C that in one form or another might appear in a TM-enabled compiler. The extensions are easy to explain.

```
void f1_1(long *r, time_t *t) {
  tm_atomic {
    *t = timestamp1;
    *r = counter1++;
  }
}

void f2_2(long *r, time_t *t) {
  tm_atomic {
    *t = timestamp2;
    *r = counter2++;
  }
}

void w1_2(long *r, time_t *t) {
  tm_atomic {
    *r = counter1++;
    if (*r & 1)
      *t = timestamp2;
  }
}

void w2_1(long *r, time_t *t) {
  tm_atomic {
    *r = counter2++;
    if (*r & 1)
      *t = timestamp1;
  }
}
```

All we have done in this case is enclose the operations within a block called tm_atomic. The tm_atomic keyword indicates that all the instructions in the following block are part of a transaction. For each of the memory accesses, the compiler could generate code as listed below. Calling functions is a challenge since the called functions also have to be transaction-aware. Therefore, it is potentially necessary to provide two versions of the compiled function: one with and one without support for transactions. In case any of the transitively called functions uses a tm_atomic block by itself, nesting has to be handled. The following is one way of doing this:

1. Check whether the same memory location is part of another transaction.
2. If yes, abort the current transaction.
3. If no, record that the current transaction referenced the memory location so that step 2 in other transactions can find it.
4. Depending on whether it is a read or write access, either (a) load the value of the memory location if the variable has not yet been modified or load it from the local storage in case it was already modified, or (b) write it into a local storage for the variable.

Step 3 can fall away if the transaction previously accessed the same memory location. For step 2 there are alternatives. Instead of aborting immediately, the transaction can be performed to the end and then the changes undone. This is called the lazy abort/lazy commit method, as opposed to the eager/eager method found in typical database transactions (described earlier in this article).

What is needed now is a definition of the work that is done when the end of the tm_atomic block is reached (i.e., the transaction is committed). This work can be described as follows:

1. If the current transaction has been aborted, reset all internal state, delay for some short period, then retry, executing the whole block.
2. Store all the values of the memory locations modified in the transaction for which the new values are placed in local storage.
3. Reset the information about the memory locations being part of a transaction.

The description is simple enough; the real problem is implementing everything efficiently. Before we discuss this, let's take a brief look at whether all this is correct and fulfills all the requirements.

CORRECTNESS AND FIDELITY

Assuming a correct implementation (of course), we are able to determine whether a memory location is currently used as part of another implementation. It does not matter whether this means read or write access. Therefore, it is easy to see that we are not ever producing inconsistent results. Only if all the memory accesses inside the tm_atomic block succeed and the transaction is not aborted will the transaction be committed. This means, however, that as far as memory access is concerned, the thread is completely alone. We have reduced the code back to the initial code without locks, which obviously is correct.

The only remaining question about correctness is: will the threads using this TM technology really terminate if they are constantly aborting each other? Showing this is certainly theoretically possible, but in this article it should be sufficient to point at a similar problem. In IP-based networking (unlike token-ring networks) all the connected machines could start sending out data at the same time. If more than one machine sends data, a conflict arises. This conflict is automatically detected and the sending attempt is restarted after a short waiting period. IP defines an exponential backup algorithm that the network stacks have to implement. Given that we live in a world dominated by IP-based networks, this approach must work fine. The results can be directly transferred over to the problem of TM.

One other question remains. Earlier we rejected the solution of using a single lock because it would prevent the concurrent execution of f1_1 and f2_2. How does it look here? As can easily be seen, the set of memory locations used for the two functions is disjunct. This means that the set of memory locations in the transactions in f1_1 and f2_2 is also disjunct, and therefore the checks for concurrent memory uses in f1_1 will never cause an abort because of the execution of f2_2 and vice versa. Thus, it is indeed trivially possible to solve the issue using TM.

Add to this the concise way of describing transactions, and it should be obvious why TM is so attractive.

WHERE IS TM TODAY?

Before everybody gets too excited about the prospects of TM, we should remember that it is still very much a topic of research. First implementations are becoming available, but we still have much to learn. The VELOX project (http://www.velox-project.eu/), for example, has as its goal a comprehensive analysis of all the places in an operating system where TM technology can be used. This extends from lock-free data structures in the operating-system kernel to high-level uses in the application server. The analysis includes TM with and without hardware support.

The VELOX project will also research the most useful semantics of the TM primitives that should be added to higher-level programming languages. In the previous example it was a simple tm_atomic keyword. This does not necessarily have to be the correct form; nor do the semantics described need to be optimal.

A number of self-contained STM implementations are available today. One possible choice for people to get experience with is TinySTM (http://tinystm.org). It provides all the primitives needed for TM while being portable, small, and depending on only a few services, which are available on the host system.

Based on TinySTM and similar implementations, we will soon see language extensions such as tm_atomic appear in compilers. Several proprietary compilers have support, and the first patches for the GNU compilers are also available (http://www.hipeac.net/node/2419). With these changes it will be possible to collect experience with the use of TM in real-world situations to find solutions to the remaining issues—and there are plenty of issues left. Here are just a few:

Recording transactions. In the preceding explanation we assumed that the exact location of each memory location used in the transaction is recorded. This might be inefficient, though, especially with HTM support. Recording information for every memory location would mean having an overhead of several words for each memory location used. As with CPU caches, which theoretically could also cache individual words, this often constitutes too high of a price. Instead, CPU caches today handle cache lines of 64 bytes at once. This would mean for a TM implementation that step 2 in our description would not have to record an additional dependency in case the memory location is in a block that is already recorded.

But this introduces problems as well. Assume that in the final example code all four variables are in the same block. This means that our assumption of f1_1 and f2_2 being independently executable is wrong. This type of block sharing leads to high abort rates. It is related to the problem of false sharing, which in this case also happens and therefore should be corrected anyway.

These false aborts, as we might want to call them, are not just a performance issue, though. Unfortunate placement of variables actually might lead to problems with them never making any progress at all because they are constantly inadvertently aborting each other. This can happen because of several different transactions going on concurrently that happen to touch the same cache memory blocks but at different addresses. If blocking is used, this is a problem that must be solved.

Handling aborts. Another detail described earlier is the way aborts are handled. What has been described is the so-called lazy abort/lazy commit method (lazy/lazy for short). Transactions continue to work even if they are already aborted, and the results of the transaction are written into the real memory location only when the entire transaction succeeds.

This is not the only possibility, though. Another possibility is the exact opposite: the eager/eager method. In this case transactions will be recognized as aborted as early as possible and restarted if necessary. The effect of store instructions will also immediately take effect. In this case the old value of the memory location has to be stored in memory local to the transaction so that, in case the transaction has to be aborted, the previous content can be restored.

There are plenty of other ways to handle the details. It might turn out that no one way is sufficient. Much will depend on the abort rate for the individual transaction. It could very well be that compilers and TM runtimes will implement multiple different ways at the same time and flip between them for individual transactions if this seems to offer an advantage.

Semantics. The semantics of the tm_atomic block (or whatever it will be in the end) have to be specified. It is necessary to integrate TM into the rest of the language semantics. For example, TM must be integrated with exception handling for C++. Other issues are the handling of nested TM regions and the treatment of local variables (they need not be part of the transaction but still have to be reset on abort).

Performance. Performance is also a major issue. Plenty of optimizations can and should be performed by the compiler, and all this needs research. There are also practical problems. If the same program code is used in contested and uncontested situations (e.g., in a single-threaded program), the overhead introduced through TM is too high. It therefore might be necessary to generate two versions of each function: one with TM support and the other without. The TM runtime then has to make sure that the version without TM support is used as frequently as possible. Failure to the one side means loss of performance; failure to the other side means the program will not run correctly.

CONCLUSION

TM promises to make parallel programming much easier. The concept of transaction is already present in many programs (from business programs to dynamic Web applications), and it has proved reasonably easy to grasp for programmers. We can see first implementations coming out now, but all are far from ready for prime time. Much research remains to be done. Q

REFERENCES

1. Herlihy, M., Moss, J.E.B. 1993. Transactional memory: Architectural support for lock-free data structures. *Proceedings of the 20th International Symposium on Computer Architecture*; http://citeseer.ist.psu.edu/herlihy93transactional.html.
2. Drepper, U. 2007. What every programmer should know about memory; http://people.redhat.com/drepper/cpumemory.pdf.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

ULRICH DREPPER is a consulting engineer at Red Hat, where he has worked for the past 12 years. He is interested in all kinds of low-level programming and has been involved with Linux for almost 15 years.