Compile time support for using Transactional Memory in C/C++ applications

Miloš Milovanović[12], Osman S. Unsal[1], Adrian Cristal[1],
Srđan Stipić[12], Ferad Zyulkyarov[12], Mateo Valero[12]

[1]Barcelona Supercomputing Center,
Barcelona, Spain
{milos.milovanovic,osman.unsal,adrian.cristal,
srdjan.stipic,ferad.zyulkyarov,mateo.valero}@bsc.es

[2]Department of Computer Architecture,
Universitat Politecnica de Catalunya,
Barcelona, Spain
mateo@ac.upc.edu

***Abstract*—Writing concurrent applications in C/C++ that use Software Transactional Memory (STM) Libraries is complicated. The programmer needs to think at the same time about the domain specific problem, about the way of use Transactional Memory library and about how to mix these two codes, which is difficult and error prone. Our idea is to create a tool that is completely transparent to the programmer, which will simplify the access to the STM libraries. The tool described in this paper performs source to source transformation of the original C/C++ code into intermediate C/C++ code which can be compiled with a standard ANSI C/C++ compiler and linked with any STM library which satisfies the defined interface. Although the tool is created for the use with STM libraries, it's object oriented design allow us to easily customize it to transform the code into the form appropriate for Hardware Transactional Memory (HTM) or Hybrid transactional memory (HyTM).**

***Index Terms*—C/C++, code transformation, code generator, STM, HTM, HyTM, OOP, OOD**

I. Introduction

The trend towards incorporating more cores in Chip Multiprocessors (CMP) will continue, with the potential for hundreds of cores for future technology generations. Inefficient data access ordering and synchronization primitives such as locking will limit programmer productivity and application performance for those future processors. Hardware or Software Transactional Memory (TM) is a crucial mechanism to tackle this problem by abstracting away the complexities associated by concurrent access to shared data [9] where multiple threads can simultaneously access a shared memory section atomically. Currently library-based Software Transactional Memory (STM) packages exist to facilitate writing TM applications.

However, writing C/C++ code which uses Software Transactional Memory (STM) libraries is very difficult. The interfaces of the current STM libraries are challenging to use for bigger applications: the code soon becomes messy because "useful" code is mixed with STM library calls and makes it difficult for development, maintenance and debugging. This defies one of the most important goals of TM, namely abstracting away the complexities of parallel programming. Another disadvantage of those libraries is that adopting old (legacy) code to use Transactional Memory is almost impossible.

To address this issue, we developed a tool that provides compile time support to make it much easier to develop TM (both STM and HTM) applications. With compile time support, C/C++ code will be the same as it was before, except the new keyword `atomic` will be introduced. This approach transforms the original C/C++ code into the intermediate C/C++ code which can then be compiled with standard ANSI C/C++ compilers and linked with any STM library which has an adapter for the required interface. The transformation is presented in the figure 1.

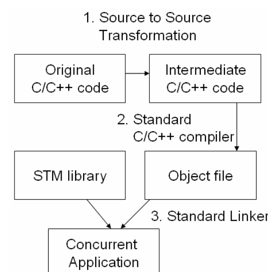

Fig. 1. Transformation process. The original code is transformed first into the intermediate code which is later compiled and linked with STM library. In this way STM is completely transparent to the programmer.

The main contributions of this paper are as follows:

- We develop a tool that makes developing TM applications using STM libraries much easier.
- The tool can be easily configured for STM, Hardware transactional memory (HTM) and Hybrid transactional memory (HyTM).
- Extend the C grammar to use a new keyword: `atomic`.
- Support in the extended grammar for nested transactions, functions, and code composition.
- Discussion for non-transactional treatment of non-shared variables for high-performance code.

II. Problem Statement

The programmer should be able to write standard C/C++ code with the only difference that he or she should use additional keyword `atomic` in order to specify that the following block of instructions should be executed atomically. STM library should be completely transparent to the programmer. In the early phases of the development or in some phases of debugging, atomic keyword should be just ignored and

ordinary C/C++ application should be produced. Current implementation of this tool transforms the code from the atomic blocks in such a way that for each memory access, a proper STM library function call is invoked. The example in the figure 2 describes the transformation.

```
atomic {      {startTransaction();
 a=a+1;         write(t, &a, *read(t, &a) +1);
 b=b-1;         write(t, &b, *read(t, &b) -1);
}              endTransaction();}
     i)                         ii)
```

Fig. 2. Example of the code transformation. In the original code (column *i*) variables *a* and *b* should be changed atomically. Transformed code is presented in the column *ii*. Macros for the start and the end of the transaction are added and each memory access is wrapped into the proper library's read/write function. Each memory access is associated with the started transaction *t*.

A useful but not mandatory feature would be that the programmer gives additional information to the compiler if some memory location is shared, private or local (private but should not be restored in case the transaction is aborted). This information can be very useful to the tool because in this case it shouldn't protect some memory locations (e.g., `local int x ;`). Generated code becomes much faster in this way because there is no indirect access to memory in some cases. Another useful information would be providing the initial read and write sets of the transaction for the STM library. This can be done manually by adding some new keywords or automatically with this tool by doing more than one pass through the code. Features mentioned in this paragraph are not implemented yet but are supported by the tool design.

III. Proposed Solution

This tool performs the source to source transformation of the C/C++ code into intermediate C/C++ code which can be compiled with standard ANSI C/C++ compiler and linked with any STM library that complies with out interface. Before we describe the tool design and behavior we will present the interface and the expected behavior of the STM library because it has significant influence on the tool design.

A. *STM library interface and behavior*

For the integration with this tool, we have implemented our own STM/TM library. A detailed explanation of the library is out of our scope; therefore we present the relevant issues here. The library satisfies the interface presented in the figure 3. Note that this interface is similar to the ones provided by current STM libraries [7], [8].

```
void   starttx(Transaction *tr);
Status committx(Transaction *tr);
void   destroytx(Transaction *tr);
void   aborttx(Transaction *tr);
void   retrytx(Transaction *tr);
void*  readtx(Transaction *tr,
            void *addr, int blockSize);
void   writetx(Transaction *tr,
            void *addr, void *obj,
            int blockSize);
```

Fig. 3. STM library interface. Figure represents the set of functions which should be implemented by any STM/HTM/HyTM in order to be integrated with the code generated by this tool. Functions are self explanatory.

The library functions have following semantics: `starttx` and `destroytx` initialize and destroy the required data structures for the execution of a transaction, `committx` publishes the results of the transaction, `aborttx` cancels the transaction and `retrytx` cancels the transaction and restarts it. `readtx` and `writetx` are function library function calls for handling memory accesses. The transaction should be started and ended with the code presented in the figure 4.

```
{ Transaction tr; Transaction* t = &tr;
  Status ret; starttx (t);
  while (1) {
      ret = setjmp (t->env);
      if (ret == TRANSACTION_STARTED){
```

Fig. 4a. Code for starting the transaction. Macro startTransaction ().

```
        If (COMMIT_SUCCESS == committx (t))
             break;
        else aborttx (t);
     } else aborttx (t);
  }
  destroytx (t);
}
```

Fig. 4b. Code for ending the transaction. Macro endTransaction ().

```
startTransaction();
// transaction body
endTransaction();
```

Fig. 4c. Code of the transaction. Using macros startTransaction() and endTransaction ().

This approach is good for the following reasons:

- Nested transactions are supported. The whole transaction is placed into a single block so that we will not have the problem with nesting the transactions (e.g., there is no name clashing for variable *t* or *tr* of the main transaction and its nested transactions).
- Code transformation would be simpler because a single macro should be placed just at the beginning and at the end of the atomic block.

1) `readtx` *and* `writetx` *functions*

The most important parts of the library are surely function calls `readtx` and `writetx` and they require special attention. Both functions operate on the byte level. `writetx` receives the real address where the data should be stored, a size of data and the data itself. The way this is done is implementation dependent. This is extremely important because different implementations can be tested without any knowledge of the rest of the environment.

Function `readtx` requires special attention because there is a possible memory leakage if the connection between the tool and the library is not established properly. Function `readtx` accepts the real address which should be read and the size of the data, and returns the pointer to the location which holds the requested data. Returned pointer does not need to be the same as the original one and this is implementation dependent. This can be the source of the memory leakage. There are two possible implementations of function `readtx`. The first implementation option can copy the data to a new location and return the pointer, requiring the programmer to free it when it is not needed any more. The other option, which we implemented, is that the library takes care about everything and returns the pointer to the location which should be just read. This pointer should not be used later for writing. If the same data should be modified later, that should be done through function `writetx` and not directly through returned pointer. In this way, internal implementation of the library is not exposed to the tool. This implementation of the library is much more suitable for the design of our tool because there is

a strict separation between the tool and the library and their responsibilities. This makes development of both tool and library much easier, flexible and portable.

B. *Tool Design*

Code transformation is divided into two phases. The idea of this separation was to support the whole C/C++ grammar outside the atomic blocks immediately. So in the first phase atomic blocks are detected and the rest of the code, outside the atomic blocks, is left without changes. In the second phase, code inside the atomic blocks is transformed as will be described in this section. Phases of the transformation are presented on the figure 5.

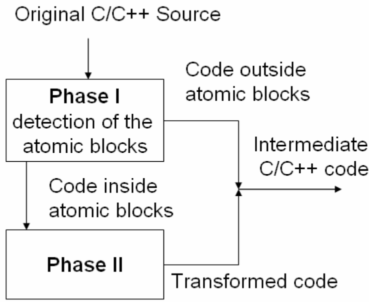

Fig. 5. Transformation process flow diagram. In the first phase, the atomic blocks are extracted from the original code. In the second phase, code from atomic blocks is transformed into the STM format. With this approach the whole C/C++ grammar is supported outside the atomic blocks instantly.

1) *Transformation process – Phase I*

In this phase code outside atomic blocks is left as is and code from the atomic blocks is extracted and passed to the further processing in the phase II. Phase I transformation is created using Lex & Yacc (Flex & Bison) [2], [3]. Currently we use the C grammar defined by Jeff Lee [4]. The C grammar is extended to support the new keyword `atomic`. The same approach can be used for C++, but the grammar should be changed.

2) *Transformation process – Phase II*

The second phase is a bit more complicated and requires more attention. If the statement is being processed in this phase, it means it is in the atomic block and needs the transactional memory access protection. Each statement is transformed into the form in which every memory access is wrapped into the proper STM library function `readtx` or `writetx`.

In this phase we also use Lex & Yacc and the analogy between Yacc grammar and inheritance in the object oriented programming. That analogy is presented and described in the figure 7.

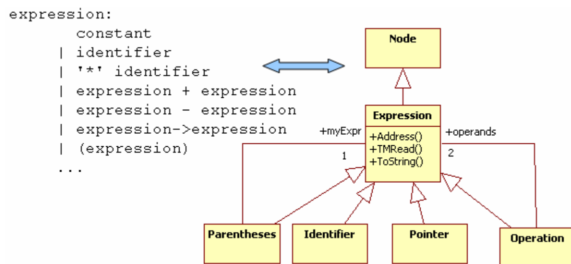

Fig. 7. Analogy between Yacc grammar and OO software design. We use the analogy between OO software and Yacc grammar to express the same concepts into two different forms. OO form can easily support a wide variety of optimizations and is much more flexible for code analysis and manipulation.

Because of this analogy, the software which performs the phase II transformation is divided into tree layers whose logical correlation is presented in figure 8.

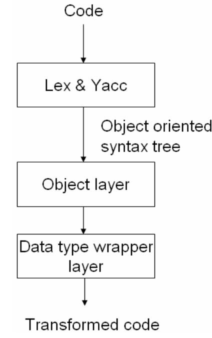

Fig. 8. Layers of the Phase II transformation. Original code will be parsed with Lex & Yacc generated parser and Object oriented syntax tree will be produced. In the object layer variety of optimizations can be performed on the given OO syntax tree and finally code should be transformed to call STM library. Data type wrapper layer is introduced to give the structure to the stream of bytes coming from STM library, because STM library operates on the byte level and is not concerned about the types.

Phase II transformation will be designed in the following way: in the Lex & Yacc tool, the grammar of the code, which can be transformed is described; for each non-terminal in the grammar, a corresponding class will be created in the object part of software. Whenever a rule needs to be reduced in the parsing process [11], and some non-terminal is produced and placed on the non-terminal stack; we will create an object of the analog class and place it on the object stack. In that way, in the end we will have the object oriented syntax tree in which each node will know how to wrap itself into the proper STM calls. An example is presented in figure 9.

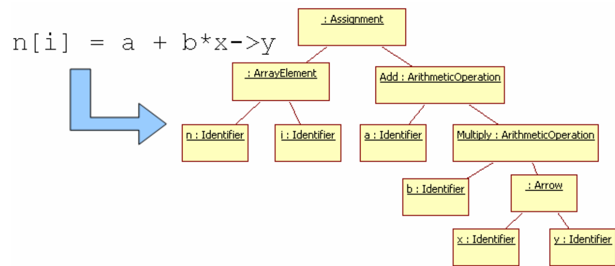

Fig. 9. Transformation from the code to the object oriented syntax tree. With this transformation each element from the code will be presented with the appropriate object. Each object will know how to wrap itself in the proper STM library function. With the simple recursion (depth first search) the whole tree will be transformed to the proper form.

Responsibilities and design of phase II layers will be presented in more detail in the following text and they will be illustrated through the transformation of the example code: `a = b + *ip + 5;`

a) *Lex & Yacc layer*

This layer is developed by changing the C grammar. In the current version we created our own C grammar; enhancing the original C/C++ grammar is future work. In the grammar, special attention is given to the assignment statement and to the expressions. In this version we also support the following statements: IF/ELSE IF/ELSE, FOR, WHILE and of course ATOMIC (for nested atomic statements). Figure 10 presents the part of the Yacc grammar which will help us to describe the transformation of the example code `a = b + *ip + 5`.

```
expression:
 CONSTANT
  { s.push(new Constant(pp1lval.value)); }
 |IDENTIFIER
   { s.push(new Identifier(pp1lval.value)); }
 |'*' IDENTIFIER
   { s.push(new Pointer(pp1lval.value)); }
 | expression '+' expression     {
      Node *left, *right;
      right = s.pop(); left = s.pop();
      s.push(new Operation('+',left,right));
   }
   …;
```

Fig. 10a. Yacc grammar for the expression non terminal. Expression can be a constant, identifier, pointer value or arithmetic operation. Whenever some rule should be reduced, appropriate object will be stored to the object stack. When a constant or identifier or pointer is reduced, the new object is placed on the stack. If the arithmetic operation should be reduced, then two last objects are popped from the stack, new arithmetic operation object is placed on the stack and previous two objects are given to the new object as its arguments.

```
assignment_statement:
  expression '=' expression {
    Node *left, *right;
    right = s.pop(); left = s.pop();
    s.push(new Assignment(left, right));  };
```

Fig. 10b. Yacc grammar for the assignment operation. Similar to figure 10a, the last two objects are removed from the stack and new assignment object is placed on the stack with two arguments: one represents the lvalue of the assignment and the other the value which should be assigned.

In case of our example the parsing process will have several steps which are presented in the figure 11.

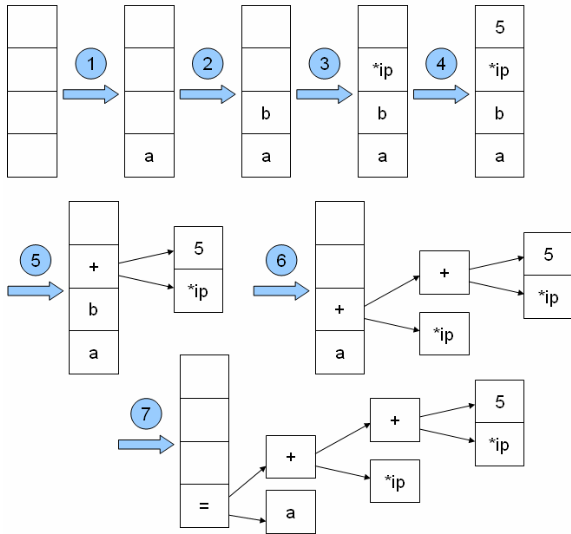

Fig. 11. Parsing of the example code `a = b + *ip + 5;` **and creation of an object oriented syntax tree.** The transformation Process is performed in following steps: 1. identifier *a* is reduced; 2. identifier *b* is reduced' 3. pointer **ip* is reduced; 4. constant *5* is reduced; 5. operation + is reduced; 6. operation + is reduced; 7. assignment is reduced.

At the end of the parsing process we will have the statement object on the end of the object stack. That statement object (the whole statement sub-tree) shall be processed further in the following object layer.

b) Object layer

This layer will perform the final transformation. The basic idea is that every object knows how to wrap itself into the appropriate STM function calls. With that property, the whole code (the whole object structure) can be transformed with simple recursion (depth first search). The big advantage of the object orientation of this layer is that it can be easily enhanced with a variety of optimizations. For example, currently each identifier object will be wrapped in function calls by default, but if there is information that it is some local data then object will not wrap itself.

Major classes from the object layer are presented in the figure 12. We have two base classes `Statement` and `Expression` and all other classes are derived from one of them.

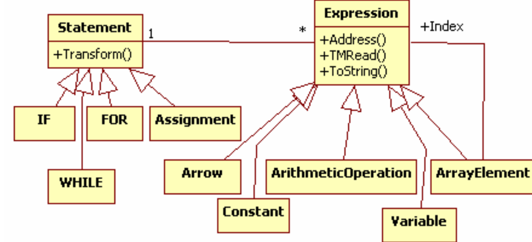

Fig. 12. Class diagram. All classes in the Object layer are derived from classes `Statement` and `Expression`. Each class will know how to wrap itself into the proper read/write functions. Virtual abstract methods `Address` and `TMRead` are responsible for that transformation and they need to be redefined in each class.

Now we will describe how this mechanism works on our example `a = b + *ip + 5`. The transformation is presented in the figure 13.

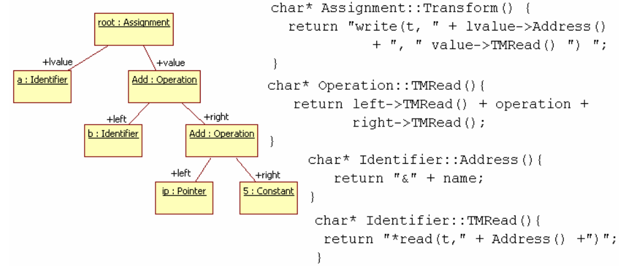

Fig. 13. Object syntax tree for the example code and its transformation. Assignment object will perform the transformation by combining the address of its lvalue sub node and the transformation of its value sub node. Then value sub node will call recursively transformation the rest of the tree. Transformation code is presented in the pseudo C++ like code.

Example code `a = b + *ip + 5` after the transformation will look like this:

```
write(t, &a, *read(t, &b) + *read(t, ip) + 5);
```

c) Data type wrapper layer

The last problem we had was the type problem. STM libraries will work on the byte level, so for example signature of the function `readtx` is:

```
void*  readtx(Transaction *tr,
              void *addr, int blockSize);
```

Currently we do the transformation on the statement level so we do not have the information about any identifier type. We have a problem when transforming code like `a+b` directly into: `*(typeof(a)*)readtx(t, &a, sizeof(a)) + *(typeof(b)*)readtx(t, &b, sizeof(b))`. The problem is that the function `typeof` is not in the C/C++ standard and is available only in `gcc`. This problem can be resolved by performing the transformation on the whole code level and by keeping track about the definition of each identifier (e.g., maintaining symbol table). This is planned for the future development in which our own `typeof` function will be developed. Currently we solve this problem by using the C++ compiler to compile the code and by adding the data type wrapper layer. For each primitive data type, we provide the function pair `read/write` which wraps the STM library

functions `readtx`/`writetx` respectively. For example for the `int` type `read` function is defined as:

```
int* read(Transaction* t, int* adr) {
  return (int*)
           readtx(t, adr, sizeof(int)); }
```

This approach can also be generalized for the user defined types (structures and typedefs) so proper read/write wrapper functions can be generated. For example if we have the structure Account, its read function can be easily generated to have the following definition:

```
Account* read(Transaction* t, Account* adr) {
  return (Account*)
           readtx(t, adr, sizeof(Account)); }
```

Currently read/write functions for user defined types are defined manually.

IV. Results

With this tool we achieved following results: *1*) Created concurrent application which operates on the Binary search trees; *2*) Supported nested transactions; *3*) Supported function calls; *4*) Supported composition of code which contains atomic statements; *5*) We successfully applied this tool to the AMMP package from the SPEC OMP 2001 benchmark [5]; *6*) Implemented support for Hardware Transactional Memory; *7*) Ideas on how the tool can be easily extended to support Hybrid Transactional Memory and produce high performance concurrent applications.

1) Concurrent application which operates on the Binary search trees

We successfully created a concurrent application which operates on the Binary search trees. The standard non-atomic functions for insertion, removal and search of (key, value) pairs in the Binary tree structure were created first. Then by simply surrounding the code inside these functions into atomic blocks, and automatic code transformation, we generated the concurrent application which uses the STM library. Sample of the original code from the insertion function and the generated code is presented in the figure 14.

```
atomic { if (*rootp == 0) { *rootp = n; }
    else {  curr = *rootp;
      while (inserted == 0) {
        if (curr->key == key){
            curr->value = value;
            curr->valid = 1;
          inserted = 1; f = 1;
        } else if (curr->key> key){
            if (curr->left == 0) {
               curr->left = n; inserted = 1;
            } else curr = curr->left;
        } else {
            if (curr->right == 0) {
                 curr->right = n;
                 inserted = 1;
            } else curr = curr->right;
        }
    } }
}
```

Fig. 14a. Original code. Part of the original non-atomic function for insertion of (key, value) pair into binary search trees.

```
{ startTransaction();
{ if (  *read(t, rootp)  == 0 ) {
       write(t, rootp,  *read(t, &n) );
}else {write(t, &curr,  *read(t, rootp) );
  while (  *read(t, &inserted)  == 0) {
   if (*read(t, &((*read(t,&curr))->key))==
      *read(t, &key)  ) {
      write(t, &((*read(t, &curr))->value),
               *read(t, &value) );
       write(t,&((*read(t,&curr))->valid),1);
       write(t, &inserted, 1);write(t, &f, 1);
    }
    else if (
      *read(t,&((*read(t, &curr))->key)) >
      *read(t, &key)  ) {
       if(*read(t,&((*read(t, &curr))->left))
            == 0 ) {
         write(t, &((*read(t, &curr))->left),
                   *read(t, &n) );
         write(t, &inserted, 1);
       } else write(t, &curr,
         *read(t,&((*read(t,&curr))->left)) );
    } else {
        if(*read(t,&((*read(t,&curr))->right))
            == 0 ) {
         write(t,&((*read(t, &curr))->right ),
                 *read(t, &n) );
         write(t, &inserted, 1);
        } else write(t, &curr,
        *read(t, &((*read(t,&curr))->right)));
    }
   }}
}
 endTransaction();
}
```

Fig. 14b. Generated code. Generated code of the insert function, which uses STM. Code is manually transformed to fit into a one column.

2) Nested transactions

We supported nested atomic blocks and the transformation to the nested transactions as illustrated with the artificial example in the figure 15.

```
void test(int k){
  atomic {
    i = i + 1;
    atomic {
      j = j + k;
      i = i - k;
    }
    p = p + 1;
  }
}
```

Fig. 15a. Original example code that contains nested atomic blocks.

```
void test(int k){
 { startTransaction();
   { write(t, &i,  *read(t, &i) +1);
      { startTransaction();
        { write(t, &j,
             *read(t, &j) + *read(t, &k) );
          write(t, &i,
             *read(t, &i) - *read(t, &k) );
         }
        endTransaction();
      }
     write(t, &p,  *read(t, &p) +1);
   }
   endTransaction();

  }
}
```

Fig. 15b. Generated code that contains nested transaction.

3) Function calls

We supported function calls for the two types of functions: *1*) Functions without side effects and which only accept arguments without pointers inside (e.g., mathematical functions which accept pure data types) and *2*) Functions which have atomic blocks inside its body. Only those two types are safe to be executed in the Transactional Memory Environment, the first one because they operate just on local variables on the stack and the second one because they already have atomic blocks to protect critical code. Figure 16 presents the transformation and usage of the function `sum`.

```
int sum(int x, int y, int* p){
  atomic { x = x + y + *p; }
  return x;
}
void useOfSum(){
  int a,b,c,d, *pc;
  a = 2; b = 3; c = 4; pc = &c;
  atomic{
    d = a + sum(a*b, a**pc);
  }
  printf(" d = %d\n", d);
}
```

Fig. 16a. Original code of the function sum and example of its usage.

```
int sum(int x, int y, int* p){
  {startTransaction();
     write(t, &x,  *read(t, &x) +
          *read(t, &y) + *read(t, p) );
   endTransaction();
  }
  return x;
}

void useOfSum(){
  int a,b,c,d, *pc;
  a = 2; b = 3; c = 4; pc = &c;
  { startTransaction();
      write(t, &d,  *read(t, &a) +
          sum(*read(t, &a) * *read(t, &b) ,
               *read(t, &a) * *read(t, pc) ));
   endTransaction();
  }
  printf(" d = %d\n", d);
}
```

Fig. 16b. Generated code for the function call sum. Function's real arguments are wrapped into the STM function calls so they will belong to the transaction read set. Body of the function sum is also transformed properly.

4) Composition of code

The most important goal we achieved is that we allow the programmer to compose the atomic based code. Composability is the essence and the biggest advantage of Transactional Memory over lock based approaches. As an example, we will use the same binary tree application. In particular, we implemented function `move`, which atomically moves a given key from the source binary tree to the destination binary tree, by composing existing atomic functions `insert`, `remove` and `search`, which were mentioned before. Figure 17 illustrates the function `move`.

```
atomic {
  if (search(*source, key, valp)) {
    val = remove(source, key);
    insert(dest, key, val);
  }
}
```

Fig. 17a. Original function move. Function move is the composition of the existing atomic functions insert, remove and search.

```
{ startTransaction();
  {if (search( *read(t, source),
       *read(t, &key) , *read(t, &valp))){
   write(t, &val,
   remove(*read(t, &source),*read(t, &key)));
   insert(*read(t, &dest) , *read(t, &key) ,
         *read(t, &val) ) ;
  }}
   endTransaction();
}
```

Fig. 17b. Generated code for function move.

5) SPEC OMP 2001 Benchmark – AMMP package

We translated the critical sections of SPEC OMP2001 benchmark [5] from the package AMMP [6]. We just surrounded the critical sections with the atomic block and translated code. Original AMMP code and its translation are presented in the figure 18.

```
atomic {
    ux = (a2->dx -a1->dx)*lambda +
          (a2->x -a1->x);
    r =one/( ux*ux + uy*uy + uz*uz);
    r0 = sqrt(r);
    ux = ux*r0;
    k = -dielectric*a1->q*a2->q*r;
    r = r*r*r;
    k = k + a1->a*a2->a*r*r0*six;
    k = k - a1->b*a2->b*r*r*r0*twelve;
    a1fx = a1fx + ux*k;
    a2->fx = a2->fx - ux*k;
}
```

Fig. 18a. Representative critical section from AMMP package. Figure presents just transformation for *x* axis. For *y* and *z* axis's is identical code.

```
{ startTransaction();
{write(t, &ux, (*read(t, &((*read(t, &a2))->dx))-
  *read(t, &( ( *read(t, &a1))->dx))) *
  *read(t,&lambda)+(*read(t,&((*read(t,&a2))->x))-
  *read(t, &( ( *read(t, &a1)) ->x))));

  write(t, &r,  *read(t, &one) /( *read(t, &ux) *
   *read(t, &ux) + *read(t, &uy) * *read(t, &uy) +
   *read(t, &uz) * *read(t, &uz) ));

  write(t, &r0, sqrt( *read(t, &r) ));
  write(t, &ux,  *read(t, &ux) * *read(t, &r0) );
  write(t, &k, - *read(t, &dielectric) *
   *read(t, &( ( *read(t, &a1)  ) ->q) ) *
   *read(t, &( ( *read(t, &a2)  ) ->q) ) *
   *read(t, &r) );

  write(t, &r,  *read(t, &r) * *read(t, &r) *
   *read(t, &r) );

  write(t, &k,  *read(t, &k) +
   *read(t, &( ( *read(t, &a1))->a) ) *
   *read(t, &( ( *read(t, &a2))->a) ) *
   *read(t, &r)* *read(t, &r0) * *read(t, &six));

  write(t, &k,  *read(t, &k) -
   *read(t, &((*read(t, &a1))->b) ) *
   *read(t, &((*read(t, &a2))->b) ) *
   *read(t, &r) * *read(t, &r) * *read(t, &r0) *
   *read(t, &twelve));

  write(t, &a1fx,  *read(t, &a1fx)+ *read(t, &ux)*
   *read(t, &k) );

  write(t, &( ( *read(t, &a2)  ) ->fx ) ,
   *read(t, &( ( *read(t, &a2)  ) ->fx) ) -
   *read(t, &ux) * *read(t, &k) );
}
  endTransaction(); }
```

Fig. 18b. Generated code for the critical section in AMMP package. Code includes transformation of the function call `sqrt`, assignment of the function return value and for variety of expression types.

6) Support for Hardware Transactional Memory

This tool is designed in such a way that can be easily customized to support Hardware Transactional Memory only by changing the `startTransaction` and the `endTransaction` macros. Those macros should be the proper hardware ISA instructions for the start and the end of the transaction. Other transformations should be just disabled and code inside atomic blocks should be left as is. All variants of Hardware Transactional Memory can be supported. Here we chose to incorporate the HTM proposed by McDonald et. al [10] with instructions `xbegin`, `xcommit` and `xvalidate` denoting the start, end and validation of the transaction respectively. In this case, the start and end macros would look like presented in figure 19. Note that endTransaction semantics require `xvalidate` since [10] uses a two phase commit.

```
#define startTransaction() \
          {  asm { xbegin }

#define endTransaction()            \
             asm { xvalidate        \
                   xcommit          \
                 }                  \
        }
```

Fig. 19. Support for HTM. Definition in pseudo code, of startTransaction and endTransaction macros in case hardware has instructions `xbegin` and `xcommit` for the start and the end of the transaction, and `xvalidate` for the validation of the transaction.

This tool is designed in an object oriented way, which allows it to be easily extended in the future with variety of optimizations directly supported in HTM. For example we noticed that not all memory accesses should be protected by the TM. Some obvious cases are: *1*) reading the stack variable, without any modification, which is very often in pure mathematical functions; *2*) variables can be declared as local (private and undo is not required) when we need the information about the commit/rollback rate. HTM instructions like `imld` and `imst` [10] for load and store without changing read and write sets directly help us to perform this optimization. In this tool each memory access is wrapped into the object and if we enhance the object with the information that it doesn't need the protection, proper code should be generated instead of the default one. Figure 20. presents this optimization on the example code `a = b + c`, where *c* doesn't need the protection, in cases of both STM and HTM.

i) `a = b + c;`

ii)
```
write(t, &a,
      *read(&b) + c);
```

iii)
```
asm {
    load  r0, b
    imld  r1, c
    add   r0, r1
    store a,  r0
}
```

Fig. 20. Example of the optimization. Transformation of the code `a = b + c`, where *c* doesn't need the protection. *i*) is the original code, *ii*) is transformation in case of STM and *iii*) is transformation in case of HTM in pseudo assembler code.

7) Support for Hybrid Transactional Memory

For the performance reasons it is easy to extend this tool to support Hybrid Transactional Memory, but at this moment we need a little help from the programmer and he/she needs to be aware of one constraint. If we use keywords `atomic` or `atomicSTM` to declare that following block should be executed atomically using STM, and add new keyword `atomicHTM` to declare that following block should be also executed atomically but using HTM, then we have support for a flavor of Hybrid Transactional Memory. `atomicSTM` blocks will be transformed into the proper STM form and `atomicHTM` in the proper HTM form. Performance of the application generated in this way will be much better because a lot of transactions are very small and can be executed using HTM which is significantly faster.

The programmer need to be aware of following constraint in order to use this kind of Hybrid Transactional Memory: the sets of memory locations which are protected by HTM and STM need to be disjoint. Currently we don't have the correlation between transactions in hardware and transactions in software and programmer needs to be aware of that. With careful program design, higher performance can be achieved compared to the case where all transactions are in software by default.

V. Previous Work

There are several implemented STM libraries [7][8]. All of them are difficult to use because the programmer need to be aware of library's API and sometimes even about the implementation details. If a bug occurs, it is extremely hard to determine if the bug is in the useful code, in the library or in their connection.

An assembler level solution named TARIFA [1] also claims to simplify writing TM code. Original code is compiled into assembler code and then TARIFA performs transformation on all memory accesses. However, one disadvantage of this approach is that it does not expose the TM to the compiler, thus missing many TM compiler optimization opportunities that are recently proposed such as the ones considered by Harris et. al [12]. Another issue is that it is not possible in TARIFA for the programmer to give hints to the compiler about some specific memory locations, like in the case of some memory location being private for the thread and therefore not requiring to be protected. Even in the private data there can be two different types: the one which should be restored on the abort of the transaction and the other which should preserve the value they have in the moment of abort (e.g., to measure the commit/abort rate). All this information should be provided to the compiler in order to produce faster code.

VI. Current Status and Future Work

In the current status of the project we support complete C grammar outside atomic blocks and subset of C grammar inside atomic blocks. Because of the current design of the data type wrapper layer we need a C++ compiler to compile the transformed code. In the near future we will support the C++ grammar. After that code transformation on the level of the whole code will be considered instead of current transformation on the statement level. With such transformations, many useful optimizations will be possible.

VII. Conclusions

This tool and the transformation it performs can be very useful for many reasons:

- It allows one to write the concurrent C/C++ applications which use STM, and have better insight what is really needed to be provided by the STM and TM in general. Writing the concurrent applications is much easier now; the programmer doesn't need to be concerned about STM implementation details, e.g. read and write sets, and doesn't need to know about them at all.
- Gain closer insight of what would be helpful for the programmer to make faster concurrent applications. For example marking variables as shared, private or local. Also support is provided that not every memory access is protected (wrapped) e.g., read access to stack locations (they are private by default) without any modification.
- To measure the average size of the transaction and have better insight of when STM is better than HTM. It also can help to measure the performance of the hybrid approach and to test a variety of hybrid approaches e.g., it can generate the code which will first try to use HTM and if it aborts several times then switch to STM.

VIII. Acknowledgements

This work is supported by the cooperation agreement between the Barcelona Supercomputing Center - National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2004-07739-C02-01 and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

IX. References

[1] M. Sußkraut and U. Muller, (2006, December 16). Providing Transparent Transactions on C/C++ Memory Locations with TARIFA [Online]. Available: http://wwwse.inf.tu-dresden.de/papers/preprint-HASE2005suesskraut.pdf

[2] J. Levine, T. Mason and D. Brown, "Lex & Yacc" 2nd ed. O'Reilly press, October 1992.

[3] (2006, December 16). Flex & Bison [Online]. Available: http://dinosaur.compilertools.net/

[4] J. Lee. (2006, December 16). ANSI C Yacc grammar [Online]. Available: http://www.lysator.liu.se/c/ANSI-C-grammar-y.html

[5] (2006, December 16). SPEC OMP 2001 Benchmark [Online]. Available: http://www.spec.org/omp/

[6] (2006, December 16). AMMP Home Page [Online]. Available: http://www.cs.gsu.edu/~cscrwh/ammp/ammp.html

[7] The Rochester Synchronization Group (2006, December 16). The Rochester Software Transactional Memory Runtime [Online]. Available:http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml

[8] University of Cambridge (2006, December 16). Practical lock-free data structures Available: http://www.cl.cam.ac.uk/research/srg/netos/lock-free/

[9] J. Larus and R. Rajwar, "Transactional Memory", Morgan Claypool, 2006.

[10] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis and K. Olukotun, "Architectural Semantics for Practical Transactional Memory" in *Proc. 33th Annu. international symposium on Computer Architecture, pp. 53-65, 2006.*

[11] A. Aho, M. Lam, R. Sethi and J. Ullman, "Compilers: Principles, Techniques, and Tools", 2nd ed. Addison-Wesley 2006.

[12] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions." In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation,* June, 2006. PLDI '06.