University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Threaded-C Release 2.0: Motivation, Description, and Rationale

Guy Tremblay†

CAPSL Technical Note 09
June 15, 2000

†Dépt. d'informatique, Université du Québec à Montréal, Québec, Canada

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

This document describes a number of fixes, changes and extensions, including possible future changes, to Threaded-C. It also briefly describes how this new version of the language evolved from the previous one and explains the rationale for the proposed modifications.

Contents

List of Figures

1 Introduction

This document describes a number of fixes, changes and extensions, including possible future changes, to Threaded-C. Except for the elements mentioned in sections 7 and 8, these features define what should become Threaded-C release 1.2, to be made publicly available in August 2000. Note that it is in part because of this time constraint that a number of interesting features (section 8) will not be included in this new release, e.g., frame-related operations, fibers as entry points with formal parameters. Before describing the proposed fixes and changes, we first briefly explain how this new version of the language evolved from the previous one, that is, the process followed to develop this revised Threaded-C.

The proposed revision of the Threaded-C language described in the following sections was obtained after a long series of discussions that occurred in the Winter and Spring 2000, while I, the author of this note, was spending a 6 months sabbatical at the dept. of ECE, University of Delaware (on leave from the Dépt. d'informatique, UQAM).

In late January, I met, on a one-to-one basis, with a number of graduate students and post-docs which had some Threaded-C programming experience. I discussed with them what they considered to be the strengths, weaknesses, good and bad features of Threaded-C — a summary of these interviews is presented in appendix A. I also had numerous individual discussions with some of the people who had the most experience with Threaded-C programming (viz., K. Theobald and N. Amaral). The documentation describing the current language as well as proposed changes was also examined extensively [4, 1, 2]. Finally, I also wrote a number of Threaded-C library modules (e.g., `PEERS_LINK`, `REDUCTION`, `ATOMIC_BLKMOV`) in order to master more thoroughly Threaded-C and to understand better what really needed to be in the language vs. what could be provided as library modules.

During Spring 2000, a number of meetings ($\approx$ 10) involving a small group of people [1] were held to discuss various suggestions and alternative propositions for modifications and extensions to Threaded-C. This was also accompanied by the exchange of a large number of emails discussing these various issues. The current document is a tentative summary/synthesis of the *somewhat* general consensus that seemed to emerge (sometimes after harsh and long discussions) from these meetings.

Sections 2 and 3 do not introduce new features *per se*. Rather, they discuss a number of features which, in the previous release, either did not work correctly or were not supported but should have been. Section 4 is the most important one, as it presents the extensions and modifications which ought to be made to Threaded-C. The next section gives a list of library modules which it might be interesting to make public — some of these modules are already available but would have to be modified based on the new language, while others would have to be developed from scratch. Section 6 briefly mentions some aspects related to the general documentation, important topic if the whole EARTH/Threaded-C is to be made publicly available. Finally, for completeness and better traceability of the process, sections 7 and 8 discuss issues which were either discussed and rejected or which, because of the August time constraint, were not addressed.

2 Bugs/features in Threaded-C that should be fixed

The following are some known bugs or awkward features of Threaded-C that will have to be fixed.

A. Because a `RETURN` instruction is now allowed to occur anywhere in a threaded function (since release 1.1), the last (textual) instruction of a threaded function should not need to be an `END_FUNCTION`

[1] G. Tremblay, G.R. Gao, K. Theobald, N. Amaral, C. Morrone and M. Butala.

or a RETURN. Thus, the error message currently produced should not occur anymore.

B. The problem with "return(TO_GLOBAL(x))" should be fixed. (Currently, this generates a compile-time error.)

C. The problem with the empty _z.c file and the flex buffer overflow should be fixed.

D. The line numbers appearing in error messages produced by the pre-processor are totally wrong in certain cases, as they refer to the line number in the intermediate form file (e.g. foo.tmp.c), not the source file (e.g., foo.c). This should be fixed, because it otherwise makes it very difficult to make sense of compilation error messages (especially when the line number refers to line 450 in a 50 lines of code source program ;-).

3 Features to fix to be more faithful to standard C

The following are a number of features which should be expected to work in Threaded-C as they exist and work in standard C and have nothing to do with multi-threading *per se*. In the previous releases of Threaded-C, they were either not permitted, not correctly and fully implemented, or available only on some implementations of the EARTH architecture and RTS but not on others (e.g., MANNA vs. Portable Threaded-C on Beowulf).

A. Initializers (initialization of a variable within its definition) should be allowed.

- Simple scalar value (a must!):

```
int x = 0;
```

- More complex initializers (if time permits):

```
char *s = "abcde";
int s[] = {10, 20, 30, 40};
```

B. Any kind of static variables (both local and global) should be allowed and handled correctly. Initializers, described in the previous item, should also be allowed for static variables.

C. Pointer arithmetic as well as the & operator should work on GLOBAL handles. For example, suppose we have the following declarations:

```
struct STRUCT {
  int a, b;
  int c[10];
};
struct STRUCT *GLOBAL pt;
```

Then, all the followings would be legal global handles:

```
pt+i
&pt[i]
&pt->a
&pt->b
&pt->c[i]
```

D. Local variable declarations should be allowed in blocks:

```
for ( ... ) {
   int r1, r2; // Local declaration
   ...
}

FIBER F {
  int r;
  ...
}
```

Implementation note: All such variables should be brought into the frame renamed in a unique way to distinguish them from other variables with the same name in different scopes.

Note that this means any such variable will be *shared* by all fiber instances. In a non-SMP implementation or in a SMP implementation where a single fiber from a given function activation can be executing at any given time, this should not cause any problem. However, this straightforward solution may not be appropriate in an SMP implementation which allows multiple instances of the same fiber to be executing at the same time; for release 1.2, it will be assumed that the program's behavior in that case is not well-defined, unless the fiber is explicitly indicated as being `EXCLUSIVE` (see section 4).

4 New features: modifications and/or extensions)

The following are modifications or extensions of Threaded-C for which there seemed to be a (*somewhat* ;-) general consensus.

4.1 Name changes

A. The keyword `THREAD` will be replaced with `FIBER` where appropriate.

Rationale: This will be more in line with the now accepted terminology, as described in [3].

B. Threaded-C instructions that do not need any argument will be written without parentheses. For example:

```
END_FIBER;
TERMINATE;
POLL;
```

Rationale: The use of explicit parentheses for these instructions was due mainly to the way the first MANNA implementation had been done and is not required anymore with the portable release.

Note that `FRAME_ADR()` is not an instruction, but an expression whose value depends on the activation. Thus, the `()` will still be required. At first, this may seem to contrast with `NODE_ID` and `NUM_NODES`; this, however, is not really the cases, because the latter can be seen as global constants, defined differently on each node.

C. The use of the "`_SYNC`" suffix will be restricted to instructions where an operation is followed by some synchronization. Thus, the name of the following two operations will be changed:

```
INIT_SYNC => INIT_SLOT

INCR_SYNC => INCR_SLOT
```

Rationale: Uniformity of naming convention.

D. To convert a slot identifier or number into an SPTR, the following operation will be used instead of SLOT_ADR:

```
SPTR TO_SPTR( LOCAL_SLOT_ID s );
```

Here, LOCAL_SLOT_ID (resp. LOCAL_FIBER_ID) is a type name that will allow a slot (resp. a fiber) to be identified either as a symbolic identifier or as an explicit slot (resp. fiber) number.

Rationale: The notion of slot address was not used or mentioned anywhere in the documentation; also, the new name is more in the style of the TO_GLOBAL and TO_LOCAL operations.

E. For symmetry (GET/PUT) the name of the DATA_SYNC operation will be changed to PUT_SYNC:

```
PUT_SYNC( T datum, T *GLOBAL dest, SLOT_ID slot )
```

Here, SLOT_ID refers either to a LOCAL_SLOT_ID (local slot identifier or explicit number) or to an SPTR.

Rationale: The name change is simply for symmetry. As for the overloading, it simplifies the name space; furthermore, this is already what is done by the compiler, as the RTS only provides operations that refer to SPTR.

F. The various communication and synchronization operations will be overloaded in terms of the type of slots being synchronized. Thus, the following operations will be available, where s is a SLOT_ID (that is, it can either a local slot number or identifier, or a remote SPTR):

```
SYNC( s );
PUT_SYNC( ..., s );
GET_SYNC( ..., s );
BLKMOV_SYNC( ..., s );

INCR_SLOT( s, ... );
```

The same type of overloading should be allowed for SPAWN and RSPAWN, so that the latter keyword can be eliminated.

Rationale: Same as for previous item.

G. Generic (overloaded) PUT/GET_SYNC will be available for any type (call by value semantics). Thus, all the followings will be valid instructions:

```
struct some_struct { .... };

some_struct x;

PUT_SYNC( 10, TO_GLOBAL(&v), DONE );
PUT_SYNC( slot, some_slot_adr, TO_SPTR(DONE) );

PUT_SYNC( x, some_struct_handle, TO_SPTR(READY) );
GET_SYNC( source, TO_GLOBAL(&x), GOT_X );
```

Rationale: Reduce the name space and simplify the interface of the data movement operations. Furthermore, the current compiler has all the information about the appropriate typing to make it possible to use these instructions for any type, not just simple scalar values.

4.2 Definition of fibers and binding of slots

A. It will be possible to specify the initialization and reset counts of a sync slot at the exact location where the associated fiber is defined. Furthermore, this will also allow the implicit declaration of the sync slot associated with a fiber when this slot is unique and has the same name as the fiber.

Rationale: This is simply an application of the *locality* principle (define things where you need them), which should help prevent the typical error of defining a new fiber and then forgetting to go back to the beginning of the program to add the appropriate `INIT_SLOT` (formerly `INIT_SYNC`) instruction.

For example, using the new syntax for fibers described below, we could have:

```
FIBER FIB <* 2, 3 *> {
// init = 2, reset = 3
   ...
}

FIBER DONE <* 1* >
// init = 1, reset = 1
   ...
```

Restriction: the values used for the initial and/or reset counts will have to be compile-time expressions, with the exception of allowing the use of `NUM_NODES` or `NODE_ID`. Thus, the following fiber definition should be accepted:

```
FIBER ALL_DONE <* 2*NUM_NODES *> {
  TERMINATE;
}
```

Rationale for the restriction: Since the expression will be evaluated in a different context from where it appears (as though it had appeared as the first instruction after the local declarations), this may create context ambiguity or errors. For example, the `x` used in fiber `F2` would have to be the one passed as argument, not the local one, which would not follow the usual scoping rules:

```
THREADED foo( int x, ... )
{
   ...
   FIBER F1 <* 1 *> {
      int x;
      ...
      FIBER F2 <* x+1 *> {
        ...
      }
   }

}
```

```
/* A spreaded loop. */
for( ... ) {
    ...
    INVOKE( ..., TO_SPTR(NEXT) );
    /* Implicit END_FIBER. */

    FIBER NEXT <* 1 *>;
    ...
}

/* A conditional fiber. */
if (! in_cache ) {
     BLKMOV_RSYNC( ..., TO_SPTR(NOW_IN_CACHE) );
     /* Implicit END_FIBER. */

     FIBER NOW_IN_CACHE <* 1 *> {
       ...
     }
}
...
```

Figure 1: Spreaded loop and conditional fiber

Rationale (use of `<*...*>` syntax): Initially, it was planned to use `<...>` instead. However, this seemed to create problem in zasym, which simply uses a scanner, not a complete parser. If we allow expression, albeit compile-time expressions, then the following should be legal:

```
FIBER FIB <* NUM_NODES < 10 ? NUM_NODES : 10 *>
```

B. A new syntax for fibers will be introduced:

```
FIBER SOME_NAME
  ...

FIBER SOME_NAME <* init, reset *>
  ...
```

Explicit opening and closing brackets are not required, although they can be indicated if desired:

```
FIBER SOME_NAME <* i, r *> {
  ...
}
```

Note that no special semantics is associated with such brackets. Instead, the special semantics is associated with the `FIBER` keyword: there is an implicit "`END_FIBER`" just *before* this keyword.

More generally, the semantics of a Threaded-C program is that the normal `C` flow of control is followed, except when a `FIBER` is encountered at which point the current fiber is terminated (implicit `END_FIBER`). Note also that there is an additional `END_FIBER` at the threaded function closing bracket, not an implicit `RETURN`.

An implication of this syntax is that *spreaded* loops or conditional fibers can be expressed as described in figure 1.

```
THREADED func( ... )
{
  /* Local variables shared by all fibers. */
  ...
  /* Instructions for the initialization fiber. */
  ...
  /* Implicit end of the initialization fiber. */

  FIBER F1 <* i1, r1 *> {
    /* Variables local to the fiber. */
    ...
    /* Fiber's instructions. */
    ...
  }

  FIBER F2 <* i2, r2 *> {
    ...
  }
  ...

  FIBER Fk <* ik, rk *> {
    ...
  }
  /* Implicit END_FIBER associated with the function's end. */
}
```

Figure 2: A function with only statically delimited fibers

However, contrary to release 1.1 of Threaded-C, falling through a fiber is *not* possible. Thus, the following two pieces of code *would not* be equivalent:

(i) Release 1.1:

```
THREAD_F1:
  ... // No END_THREAD => Falls through into code for THREAD_F2
THREAD_F2:
  ...
```

(ii) Release 1.2

```
FIBER F1
  ... // Implicit END_FIBER => No fall through
FIBER F2
  ...
```

The use of "{...}" can be interesting when all fibers are defined in a *static* way, that is, when all fibers are associated with a statically (textually) delimited piece of code with a single entry and a single exit point. Threaded functions written in this way will often have the form described in fig. 2, which makes the threaded function almost looks like a class. Note that another important advantage of the use of the "{...}" syntax is that the resulting program will be correctly indented using `emacs`' C-mode. This is not the case if the `FIBER` keyword and fiber name are followed by a sequence of instructions not enclosed within a block.

Rationale: Although making the END_FIBER implicit was generally accepted, this was still the issue that gave rise to the strongest debate. The two extreme positions were:

(i) Fibers should be defined strictly by statically delimited piece of code, with single entry/single exit. This was claimed to be more the spirit of the atomic fiber model of EARTH. It was also thought to be a better basis for efficient compiler optimizations.

(ii) Fibers should be defined by any flow of control sequence of code. This is more general and, furthermore, already supported by the current compiler.

The adopted position is somewhat in between, allowing some useful fibers which are not strictly statically defined piece of code (spreaded loops and conditional fibers), yet disallowing fibers with *unclean* and strange behavior (e.g, falling-through a fiber because of the absence of an END_FIBER). Note that the current compiler also allows jumping directly into another fiber. However, this will not be documented because this type of programming should not be encouraged.

C. When using symbolic fiber and slot names, the name of the slot is generally the same as the name of the fiber. Thus, the latter will be made optional in the INIT_SLOT instruction by allowing overloaded forms of INIT_SLOT with 2, 3 or 4 arguments:

```
INIT_SLOT( DONE, 1 );         // = INIT_SLOT( DONE, 1, 1, DONE );
INIT_SLOT( DONE, 1, 2 );      // = INIT_SLOT( DONE, 1, 2, DONE );
INIT_SLOT( DONE, 1, 2, FIB ); // => slot and fiber are different
```

Rationale: Support the common case (slot name = fiber name, init count = reset count) yet make it possible to express the more general case.

D. It will be legal to mix symbolic fiber and slot names with explicit fiber and slot numbers. For example, the following program would be legal:

```
INIT_SLOT( SLOT1, 1, 1, 2 );
INIT_SLOT( 2, 1, 1, FIBER1 );
INIT_SLOT( 1, 1, 1, FIBER2 );
INIT_SLOT( DONE, 1, 1, DONE );
...
FIBER FIBER1 ...  // Associated with slot 2
FIBER 2      ...  // Associated with slot SLOT1 (= 0)
FIBER FIBER2 ...  // Associated with slot 1
FIBER DONE   ...  // Associated with slot DONE (= 3)
```

Note, however, that there is a restriction: an explicit slot number must fall within the range allowed for the function. More precisely, if the threaded function introduces n slots, then an explicit slot number k must be smaller than n ($0 \leq k < n$). A new slot is introduced in two different ways:

(i) Explicitly, using an INIT_SLOT instruction. For example, these two instructions introduce two new slots:

```
INIT_SLOT( 0, ... );
INIT_SLOT( SLOT_NAME, ... )
```

(ii) Implicitly, using a FIBER declaration with an associated initial/reset count. For example, the instructions below implicitly introduce 2 slots named F1 and F2; note that the slot for F3 must be defined somewhere else in the program using an INIT_SLOT operation, since no init/reset count is indicated:

```
FIBER F1 <* 1 *>    ...

FIBER F2 <* 2, 3 *> ...

FIBER F3            ...
```

It is an error, which should be signaled by the compiler, for a program to contain both an implicit and an explicit slot definition for the same slot. For example, this would not be legal:

```
INIT_SLOT( DONE, 1 );
...
FIBER DONE <* 1 *> ...
```

Rationale: This is already correctly handled by the existing compiler. Furthermore, some people want to make it possible to use explicit slot numbers, in order to use some programming *tricks* that require playing around with the elements of the `SYNC_SLOTS` array.

E. Two new operations will be available to perform lower-level manipulations of sync slots:

`SPTR SYNC_SLOTS_FRAME(void)`
: Returns the base address of the array of sync slots for the current threaded function.

`int SLOT_OFFSET(LOCAL_SLOT_ID s)`
: Returns the offset, relative to the local sync slots array, of the local slot identified by `s`. This operation, combined with the previous one, allows a programmer [2] to do various programming *tricks* with the sync slots array elements.

4.3 Termination of fibers and threaded functions

A. The termination of a fiber can be implicit. However, the `END_FIBER` can also be indicated explicitly.

B. Considering that terminating a threaded function does not return to any specific place (unless it has been `CALL`ed) and does not return any result (`THREADED` = `void`), the instruction `TERMINATE` will now be used to indicate the termination of a function invocation, that is, to deallocate its frame.

C. The distinction between returning from an invoked threaded function (`END_FUNCTION()`) vs. from a called one (`RETURN()`) will not be made anymore. (The two keywords can be used interchangeably since release 1.1.)

D. If a function does not contain a `TERMINATE` instruction, a warning message should be produced by the compiler.

E. There is no implicit `TERMINATE` instruction at the end of a function. Thus, if a threaded function falls through its last instruction, this is equivalent to terminating the current fiber (`END_FIBER`).

F. It is an error to execute a `TERMINATE` instruction while some fibers are still active or ready to execute (in the ready queue). The exact behavior when such an error occurs depends on the implementation.

[2] Namely, Kevin ;-)

4.4 Mutual exclusion and critical sections

A. It will be made explicit, in the language reference manual, that a slot's count is reset (almost) immediately after its count drops to 0, *not* when the fiber terminates. More generally, any synchronization operation is sure to be performed atomically on the appropriate slot, including the reset, and the reset will have been done when the activated fiber has been put in the RQ.

   Rationale: This will make simpler the semantics of decrementing synchronization count and will not lead to any strange behavior if a `SYNC` arrives before the associated slot has been reset. This is already what is written down in the existing documentation (memo 19) and what seems to be assumed by most people. Furthermore, this is what all current implementations do! Of course, the upcoming implementation must continue to have this same behavior.

B. For ordinary fibers, whether a unique fiber or multiple fibers from a given function activation can be executing at any given time is implementation dependent. A program whose behavior depends on the fact that a single (ordinary) fiber from a function activation is executing at any given time may not be well defined (see next item for the `EXCLUSIVE` annotation).

   However, note that even though multiple fibers from the same function can be executing concurrently, the behavior of a program containing two or more instances of the *same* fiber executing concurrently is not well-defined as far as local variables are concerned. Thus, the exact behavior of the following program may not be well defined if fiber `F1` updates `x` or `y`:

```
SYNC( F1 );
SYNC( F1 );

FIBER F1 <* 1 *> {
   int x, y;
   ...
}
```

   Whether or not multiple instances of `F1` are allowed to execute concurrently is implementation dependent. Furthermore, if the implementation allows multiple instances to execute concurrently, whether or not the local variables of the fiber are shared is also implementation dependent.

   Rationale: The upcoming SMP implementation by C. Morrone will allow at most a single fiber from a given function activation to be executing at any given time, because each processor will have its own independent Ready Queue. However, there is no reason why this constraint should continue to hold in subsequent implementations; for example, there could be a unique RQ shared by all processors of a given node.

C. The annotation `EXCLUSIVE` will be provided to indicate that a fiber should always be the only fiber of the function to be executing. More precisely, two `EXCLUSIVE` fibers of the same function activation will not be allowed to run at the same time; on the other hand, a non-`EXCLUSIVE` fiber can be running at the same time as an `EXCLUSIVE` one.

   Note that the use of `EXCLUSIVE` relates with the immediate reset described earlier; for example, the semantics of the program presented in fig. 3 would now be well-defined: even though `F` can have 3 instances in the RQ, at most one would be executing at any given time, so the program would print "`x = 3`".

   Remark concerning spreaded loops: A typical spreaded loop such as the following can be executed without the need for any `EXCLUSIVE` annotation:

```
THREADED MAIN()
{
  int x = 0;
  SYNC( F );
  SPAWN( F );
  SYNC( F );

  EXCLUSIVE FIBER F <* 1 *> {
   x += 1;
   SYNC( DONE );
  }

  FIBER DONE <* 3 *> {
    printf( "x = %d\n", x );
    TERMINATE;
  }
}
```

Figure 3: Multiple (non-concurrent) activations of an exclusive fiber

```
for( i = 0; i < N; i++ ) {
  ...
  INVOKE( ..., SLOT_ADR(NEXT) );

  FIBER NEXT <* 1 *>;
    ...
}
```

This is so because the activation of the NEXT fiber is done from within the loop, so that the loop essentially gets sequentialized.

Rationale: The EXCLUSIVE annotation is required to make it possible to implement critical sections and mutual exclusion using Threaded-C threaded functions. In previous documentation of Threaded-C [4], the implementation of mutual exclusion rested on the assumption that at most a single fiber would be executing on any given node, which will definitively not be true in an SMP implementation. The EXCLUSIVE annotation should be seen as similar to the synchronized keyword of Java.

D. A new primitive data type (language and RTS supported) will be introduced as a mechanism to help support critical sections in a data flow style: atomic MAILBOX. The operations for this new type are presented in fig. 4. This type will allow multiple producers to send their data, in a non-interfering way, to a single "*merge point*" without any danger of them overwriting each other's data, as might occur if multiple BLKMOV_RSYNC were concurrently targeted to the same destination address.

Rationale: This is also another issue which gave rise to a lot of discussion. The reasons for choosing this mechanism over some of the other mechanisms which were discussed can be explained as follows:

- The atomic mailboxes seem quite close to the non-deterministic merge useful in data flow style programming but difficult so far to implement using an argument-fetching approach:

```
void INIT_MAILBOX( MAILBOX *mb, SLOT_ID item_dropped );
void INIT_MAILBOX( MAILBOX *mb, SPTR item_dropped );
/* The slot item_dropped will be signaled when an item is dropped
   in the mailbox and it reaches the destination mailbox. */

void DROP_IN     ( MAILBOX *GLOBAL mb,
                   void* source,
                   int size );
void DROP_IN_SYNC( MAILBOX *GLOBAL mb,
                   void *GLOBAL source,
                   int size,
                   SLOT_ID source_free );
void DROP_IN_SYNC( MAILBOX *GLOBAL mb,
                   void *GLOBAL source,
                   int size,
                   SPTR source_free );
/* Copies the data (size bytes) from source into the mailbox (global
   handle!).  Signals source_free when the source buffer can be
   safely overwritten. */

int RETRIEVE_ITEM( MAILBOX mb, void *dest );
/* An item is retrieved (and removed from the mailbox) and copied in
   (the buffer indicated by) dest.  Returns the size (number of
   bytes) for the retrieved item.  Returns 0 if the mailbox was
   empty. */

int RETRIEVE_ITEM_ADDR( MAILBOX mb, void **dest );
/* Similar to RETRIEVE_ITEM, except that it copies the address of a
   buffer containing the retrieved item in *dest. Thus, this
   operation should be used when the size of the item is not known
   beforehand. */

void FREE_MAILBOX( MAILBOX *mb );
/* Destroys the indicated mailbox. Sets *mb to NULL. */
```

Figure 4: The new primitive data type `MAILBOX` with its associated operations

the mailbox allows the merging not only of signals but also of data tokens, which are put in the mailbox by the various producers and taken out, when informed about their arrival (see the signal bound to the mailbox when it is initialized), by a consumer. Note that there is no implied FIFO property associated with the retrieval of the elements; however, the implementation may choose, for efficiency and simplicity reasons, to implement those mailboxes using straightforward FIFO queues.

- The atomic `BLKMOV_SYNC` [3] with delayed reset has a somewhat complex semantics, requiring the RTS to deal with special cases when resetting a slot and when getting ready to perform certain move operations — for example, a data move operation may have to be delayed and remain in the SU space/queue until the target slot is explicitly reset. Furthermore, this also creates two different classes of slots, with different reset behavior, and when a delayed reset slot is finally reset, it must also be determined if there is any pending block move associated with that slot.

- Split-phase locks do not go well with the data flow style and may require a larger number of remote operations when used to implement certain types of atomic updates (the lock first has to be grabbed before anything else can be done). But note that such locks can easily be implemented using the atomic queues.

Using the atomic mailboxes, it is straightforward to implement split-phase locks or linear reduction boxes in library modules (modules `LOCK` and `REDUCTION`). Atomic block moves with delayed resets can also be implemented in this way, although their implementation is more complex (a mailbox and additional locks are required: see library module `ATOMIC_BLKMOV`).

Note concerning the choice of the name `MAILBOX`: The initial name proposed for this module by the author was atomic `QUEUE`, name which was later changed to `MAILBOX` in order to eliminate the FIFO connotation generally associated with queues. The name `MAILBOX` seemed to have a nice analogy with regular (non-email) mailboxes: you can send items to a mailbox from anywhere (`DROP_IN_SYNC` can be used remotely); when an item arrives, you can know about it only locally (you can hear the mailman delivering stuff in your mailbox); you must be in your home (local) to retrieve items from your mailbox. Other names which were examined were `BAG` and `MERGE_BUFFER`.

Note concerning the various forms of `INIT_MAILBOX` and `DROP_IN_SYNC`: In order to be consistent with the other EARTH-like operations, it should be possible to use either a local or remote sync slot, when such a slot is required.

As for the simple `DROP_IN`, notices that it receives a local pointer: we can then assume that when execution proceeds to the instruction following `DROP_IN`, the item being dropped as definitively been copied into the RTS space. In turn, this means there is no need for an addition `source_free` signal.

4.5 Varia

A. Debugging support allowing the generation of a trace of all Threaded-C instructions being executed will be made available. [3] Two kinds of debugging trace will be available, activated using appropriate compile-time switches:

   (i) Level 0: Traces the entrance to and exit from used program fibers. Activated using "`-debug 0`".

   (ii) Level 1: Traces the execution of EARTH operations performed by the user program, including entrance to and exit from threaded functions. Activated using "`-debug 1`".

   Both kinds of traces can also be combined into a single trace using "`-debug 01`".

B. Generic timing instructions will be made available. The new data type and its associated operations are presented in fig. 5; the operations will be described in more detail in the language reference manual.

C. An additional overloaded `BLKMOV_SYNC` operation that allows two slots to be signaled (slot number/identifier and/or remote `SPTR`) will be introduced:

   (i) A slot to signal when the source buffer can be safely overwritten (generally local).

   (ii) A slot to signal when the content has been written to its final destination (often remote).

   A typical use of this instruction might look as follows:

[3] This has been developed as a course project by J.P. Durbano & J.A. Six during Spring 2000.

```
Data type:
  EARTH_TIME

Constants:
  EARTH_TIME_ZERO
  EARTH_TIME_RES
  EARTH_TIME_MAX
  EARTH_TIME_MAX_LONG
  EARTH_TIME_LONG

Functions
  EARTH_TIME EARTH_TIME_READ();

  EARTH_TIME EARTH_TIME_ADD(EARTH_TIME, EARTH_TIME);
  EARTH_TIME EARTH_TIME_SUB(EARTH_TIME, EARTH_TIME);

  double EARTH_TIME_NSEC(EARTH_TIME);
  double EARTH_TIME_USEC(EARTH_TIME);
  double EARTH_TIME_MSEC(EARTH_TIME);
  double EARTH_TIME_SEC (EARTH_TIME);

  void EARTH_TIME_UPDATE();
```

Figure 5: Timing related data type and operations

```
THREADED foo( SomeType *GLOBAL dest, SPTR dest_ready )
{
  SomeType source;
  ...
  BLKMOV_SYNC( TO_GLOBAL(&source), dest, sizeof(SomeType),
               SOURCE_FREE,   // The variable source can now be overwritten.
               dest_ready );  // The information has been written in dest.
     ...
  FIBER SOURCE_FREE <* 1 *>
     ...
}
```

D. Indexed set of fibers can be defined using the following syntax:

```
FIBER FIB[i: 0..N-1] {
   ... instructions that can (but need not) use i ...
}
```

The expressions e_l and e_h used as bounds in `[i:` e_l`..`e_h`]` must be compile-time expressions (although expressions using `NUM_NODES` should also be allowed).

A compile-time error should be signaled if $e_l > e_h$.

Implementation note: this should not lead to the creation of `N` clones (what if `N = 10000`?). Instead, a unique fiber, parameterized by the index, should be defined.

Suppose such indexed fibers are defined with local variables and also use some non-local variables:

```
int a, b, c;

FIBER F[i: 0..N-1] <* 1 *> {
   int x, y;      // The only two local variables of F[i].
   ... x ... y ... a ... b ...
}
```

Then:

- The non-local variables (`a, b` and `c`) should be shared between the different instances of the indexed fibers.
- Each instance with distinct index (`F[j]` and `F[k]` with `j <> k`) should have its own *private* copy of the local variables (`x` and `y`).
  Implementation hint: Such local variables could be lifted into the frame, but as an array of such variables. For example, the above example could be transformed into the following:

```
int a, b, c;
int x[N], y[N];

FIBER F[i: 0..N-1] <* 1 *> {
   ... x[i] ... y[i] ... a ... b ...
}
```

Rationale: There are programs where it is useful to have the same piece of fiber code executed, but on different parts of an array. In the old version of Threaded-C, this required having clones (almost identical copies of the same code).

E. Input/output will continue to be done using the standard C operations. When a program is running on a number of different nodes, each node will have its own private `stdout` stream. For the upcoming SMP implementation, the default will be for the output of all processors on a given node to be merged onto a common `stdout` stream. A compiler option, however, will also allow the program output produced by each processor to be sent to an output stream unique to each processor (probably called `EARTH_OUT`.

F. In the previous documentation, it was assumed that global variables were allocated in such a way that the address of any copy of a global variable would be the same on all nodes. This assumption will be dropped and this will not be discussed in the documentation.

   Rationale: This was considered an implementation-dependent feature. We could easily imagine some heterogeneous machine where, although each node has an identical copy of the code, the code and data images are not necessarily identical.

G. EARTH operations will now be allowed in ordinary sequential functions. The `CALL` instruction will be preserved.

   The following keywords or operations will not be allowed in non-`THREADED` functions: `FIBER`, `END_FIBER`, `TERMINATE`, `FRAME_ADR`, `IP_ADR`.

   Furthermore, operations or overloaded variants of operations which involve an implicit reference to the sync slots array (including any use of a `SLOT_ID`) will not be allowed either: `SYNC_SLOTS_BASE`, `SLOT_OFFSET`, `INIT_SLOT`, `TO_SPTR`, `INCR_SLOT`, `SYNC`, `SPAWN`, `PUT_SYNC`, `GET_SYNC`, `BLKMOV_SYNC`, `INIT_MAILBOX`, `DROP_IN_SYNC`. Of course, the variants that use `SPTR` will be allowed.

   A `CALL` statement will be described as a kind of synchronous (not "sequential") function invocation, where execution of the initial fiber of the callee does not require fiber switching. However, if a `CALL` instruction occurs in an exclusive fiber *and* the callee contains multiple fibers, then one should not expect the continuation code (after the return from the CALL) to still be part of the critical section.

Rationale:

- Conceptually, there is no need to prevent ordinary functions to perform EARTH operations. A function should be made THREADED only because it consists of multiple fibers, not because it uses EARTH-related operations.
- Sean Ryan's project (location consistent software cache) seems to require the use of the CALL statement: it is useful for (procedural) abstraction purpose, since it makes it possible to hide a split-phase transaction inside another function. If the CALL was not available, then the software cache management code would have to be inlined into the caller's code, which would have to be done by Sean's pre-processor. However, apart from leading to code size growth, this inlined code might contain local variables. Although these are supposed to be implemented for August's release, there is still some risk (time-related and technical problems) that they might not be available.
- Keeping the CALL instruction does not mean that the description of the semantics of the CALL statement should be kept the same: the notion of sequential execution does not seem to be appropriate, since the callee may itself contain fibers — note that those seem to be handled correctly, as the frame is indeed allocated from the heap. A CALL statement should probably be interpreted as doing a kind "synchronous" (or "blocking") call of a threaded function.
- At some point, there seemed to be a problem concerning the interaction between CALL instructions and exclusive fibers. If a CALL is done in an exclusive fiber, it might be difficult to ensure that the instructions executed after the call are still part of the initial exclusive fiber. If the CALLed function consists of a unique fiber, then there is no such problem case. If it contains multiple fibers, then the return may be done as part of a different fiber. This should be described in the documentation as an expected feature, not as a bug:
  - If you do a regular call to an ordinary function that do EARTH operations, then you do not switch fiber. (And then the problem mentioned for the implementation of I-structures disappear.)
  - If you do a CALL to a threaded function with a single fiber, then you will still be part of the exclusive fiber.
  - If you do a CALL to a threaded function with multiple fibers, then the return may not be part of the initial exclusive fiber. Thus, it makes sense that mutual exclusion has been lost.

  In conclusion, the above mean that exclusive fibers still come for free in the upcoming SMP implementation and that CALLs to functions with multiple fibers should not be expected to still be in the critical section, which is reasonable if you don't interpret a CALL as a sequential call — this was probably the real problem!
- At a certain point, it was considered to remove CALL, since THREADED functions which are CALLed often consist of a single fiber. Such functions are often introduced only because a function is required for modularity purpose, but this function cannot be a regular C function because it uses some EARTH operations. Thus, the program has to end up using a THREADED function which has a unique fiber (its initialization fiber). This special case can now be implemented using a regular call to an ordinary sequential function that performs EARTH operations.

H. In the upcoming SMP implementation, the value returned by NODE_ID will in fact be the unique number identifying the processor (which could also be called the EARTH_NODE_ID, PROC_ID or VIRTUAL_NODE_ID). Thus, if a physical node contains many processors, each of these will have *distinct* NODE_ID.

Note that the following equations will remain valid:

```
i)    OWNER_OF( MAKE_GPTR(pt, n) ) = n

ii)   TO_LOCAL( MAKE_GPTR(pt, n) ) = pt

iii)  TO_GLOBAL(pt) = MAKE_GPTR( pt, NODE_ID )
```

A corollary of the above is that, given a global handle gh, the associated address may be one that refers to local memory even though OWNER_OF(gh) != NODE_ID. Thus, the old IS_OWNER() operation should not be used anymore and will not be supported.

To determine whether the address associated with gh is really from the local memory, one should instead use IS_LOCAL(gh).

To determine if two global handles refer to the same memory module/space, the following operation acting upon node ids will be introduced:

```
int SHARE_MEMORY( int n1, int n2 )
```

Then, IS_LOCAL() would be defined as follows:

```
IS_LOCAL(gh) = SHARE_MEMORY( OWNER_OF(gh), NODE_ID )
```

Rationale: It seems that, a while ago, a design decision was made that each processor would have its own ready queue and token queue, in order to avoid having concurrent conflicting accesses to these queues by independent processors. The IS_LOCAL() operation alleviates the problem of OWNER_OF() not being valid anymore. The new SHARE_MEMORY() operation also make it possible for the user to determine in an abstract way if two nodes share the same memory.

Also note that the reason for each processor having its own id is that for-loops such as the following can still be used to invoke a given function on all the processors (virtual nodes):

```
for( i = 0; i < NUM_NODES; i++ ) INVOKE( i, foo, ... );
```

5 Library modules that will be made publicly available

This section lists a number of library modules which should be made available as part of Threaded-C release 1.2. These modules should be briefly described in the document for this new release.

A. Below are some already implemented modules (in whole or in part):

- I_STRUCTURES. The interface of this module may have to be simplified, and its implementation may have to be modified to make it *thread-safe* for the new SMP implementation.
- LPTR/LOCK. Split-phase locks.
- PEERS_LINK. Abstract data type to facilitate the setting up of producer/consumer relationships.

- Linear dynamic (unknown number of producers) and static (number of producers known beforehand with tree-based reduction) `REDUCTION` boxes. The linear reduction boxes module developed by G. Tremblay should be modified to allow for parallel multi-nodes reduction. The interface to the static tree-based reduction boxes modules developed by K. Theobald should be clarified and simplified.
- `HUB`. Synchronization point for multiple producers/consumers.
- `ACQUIRE/RELEASE` on arbitrary locations.
- Broadcasting.

B. Some other modules that could/might be implemented:

- VHDL style sensitivity list: execution of a fiber is triggered as soon as some subset of inputs is changed (Kevin's suggestion).
- Channel data type for interprocess communication (with split-phase `SEND/RECEIVE`)?
- Monitor like construct (with conditional variable)?
- `MTAKE/MPUT` operations *à la* `M-structures`.

6 Documentation

In general, both the internal and external documentation should be improved, if the whole EARTH/Threaded-C system is to be made public and open source.

A. Improve the internal documentation (comments in the code).

B. Improve the external documentation (user manual):

- Describe what should be considered good and efficient Threaded-C programming style.
- Describe the less often used features, e.g., loops spread over fibers.
- Describe the implementation-related restrictions: all fibers of same activation executed on same processor.
- Describe the possible future extensions to the language.
- Include an appendix that presents examples illustrating some of the language constructs and libraries.

C. The library modules should be documented using `DOC++`. The same could be done for all the EARTH operations.

7 Some issues that were discussed but that will not be included in Release 1.2

A. Higher-priority fibers.

The introduction of a `CRITICAL` annotation to indicate that a fiber should be scheduled with higher-priority has been rejected. It was thought that this might lead to strange (and possibly unfair) scheduling problems. For example, there could be a loop generated because some critical fibers are scheduled with higher-priority preventing all other fibers to make any progress (starvation).

However, it is not clear to the author that this is really the case: if the program goes into a loop because some fibers are scheduled with higher-priority, the same program would have gone into a loop anyway, although not as rapidly.

Note that there do exist previous dataflow systems (e.g., work done in Manchester) that do use both LIFO and FIFO scheduling depending on various program conditions. Given that in the upcoming implementation, scheduling will be done using a FIFO implementation of the RQ, it should be easy to support high-priority fibers simply by switching to LIFO scheduling for fibers which have been explicitly identified as critical by the programmer. Also note that the FIFO strategy used for the RQ is not part of the execution model *per se*. All the execution model has to ensure is that fibers in the RQ will get executed, but the exact order is not specified.

B. Split-phase locks and the `ATOMIC_BLKMOV` instruction. See discussion on atomic `MAILBOX` in section 4 (p. 12).

C. Frame-based approach: some people suggested that Threaded-C should go to a frame-based approach to handling sync slots, fibers and data exchange, instead of using an approach based on `GLOBAL` handles and `SPTR`. An important motivation seemed to be that this was in better correspondence with the underlying execution model. However, there are also a number of problems with such an approach:

- It does not seem to fit well with the standard, pointer-based, C programming style. This is especially true in regards to dynamically allocated (heap-based) objects, for which it was far from clear how a frame-based approach would handle them. Using a frame-based approach, it seemed that modifying an existing divide-and-conquer program to make it parallel using an owner-based approach would require changes not only to the function and procedures but also important changes to the data structures themselves (more precisely, adding a `GLOBAL` annotation to the pointers would not be sufficient, because these would now have to be represented in terms of frame/offset).

  It must also be noted that, although the use of the `GLOBAL` annotation for pointers may sometimes appear to be annoying (see section A), it seems to be the price to pay in order that Threaded-C programs keep control over the data movement and communication: the `GLOBAL` annotation simply reflects the dichotomy between local vs. remote data. Dropping this distinction and having all communications occur implicitly would mean the loss of control over this aspect of the computation process, which is not in the spirit of Threaded-C.
- Based on the description given in [3], it seemed that using such an approach would increase the number of parameters that would have to be transmitted both to threaded functions (thus increasing the coupling) and to primitive data movement operations (thus making them more complex to use).

8 Features that could be part of future extensions

A. Indirect `BLKMOV_RSYNC` instruction, where the source and target addresses are given indirectly. Such an instruction could be defined somewhat as follows:

```
IND_BLKMOV_RSYNC( a1, a2, n, s );
==
BLKMOV_RSYNC( CONTENT_OF(a1), CONTENT_OF(a2), n, s );
```

B. Fibers with formal arguments (*à la* token flow) and with associated `SEND` instructions. Examples are presented in appendix B. According to the author's opinion, this is probably the *most* important extension that should be made to Threaded-C to really make it a clean language. As it currently stands, Threaded-C is a dataflow inspired imperative programming language, but where the flow of data is very difficult to express in a clean way, which make most programs very difficult to understand. The reason why this is so is because coupling between fibers in distinct threaded functions has to be created through what is called *content* coupling (aka. *pathological* coupling). [4] That is, for a fiber `SEND` in threaded function `sender()` to be able to send data to fiber `RECEIVE` in function `receiver()`, the latter must first send to the former the exact location (and sync slot) into which the data will have to be moved (and the slot which will have to be signaled). This location, in many cases, will refer to some space local to the `receiver` function, which means the `producer()` routine has direct access to the inside of the `consumer()` routine, thus creating content (= strong, thus bad) coupling.

   It is important to note that implementing entry points with formal arguments is (most probably?) essentially a front-end compiler problem, involving the correct treatment of the typing information. Implementing the token flow and required buffering should probably be quite straightforward to implement as far as the RTS is concerned.

C. The approach described in the previous item (entry points with formal arguments) might not be the only possible approach to adding formal arguments to fibers. An alternative approach, somewhat in the spirit of a frame-based approach emulating an object-based, might be to make it possible to describe an explicit public interface of a threaded function in terms of the fibers and data it explicitly exports. For example, it might be possible for a fiber to indicate that some slot/fiber is to be `PUBLIC`, thus could be available given an appropriate reference to the frame of the associated function activation:

```
THREADED proc( ... )
{
  ...
  PUBLIC FIBER F0 {
     ...
  }
}

THREADED other_proc( ... )
{

  // ASSUMPTION: proc_tok is the frame id for a specific
  //             invocation of proc

  SPTR handle = MK_SPTR( proc_tok, proc::HANDLE_INFO );

  SYNC( handle );
}
```

   If such an approach was used, there would remain a number of issues to be resolved:

   - Should variables be allowed to be public (leading to content coupling)?

[4] Coupling is the *strength* of the links that exist between two modules. When coupling is strong, this means changes in one module may adversely affect the other, thus making programs difficult to maintain. The weakest and preferred form of coupling is called *data coupling*, which is when two modules only exchange information through their argument lists.

```
#include <stdio.h>
#include <stdlib.h>

THREADED int fib( int n, SPTR done )
{
  int r1, r2;

  if (n <= 1) {
    r1 = 0; r2 = 1;
    SPAWN( READY );
  } else {
    r1 := TOKEN( fib, n-1, TO_SPTR(READY) );
    r2 := TOKEN( fib, n-2, TO_SPTR(READY) );
  }

  FIBER READY <* 2 *> {
    RETURN( r1+r2, done );
  }
}

THREADED void MAIN( int argc, char* argv[] )
{
  int n, res;

  n = atoi(argv[1]);
  res := TOKEN( fib, n, TO_SPTR(FIB_DONE) );

  FIBER FIB_DONE <* 1 *> {
    printf( "fib(%d) = %d\n", n, res );
    TERMINATE;
  }
}
```

Figure 6: A THREADED Function That Returns a Result Using a Split-phase Function-call Assignment Operation

- How would one obtain the reference to the frame of a function invocation?
  - The frame reference is returned like any other value, that is, through explicit sending of information.
  - The frame reference is returned by using a special function call. In this case, either an immediate (virtual) frame reference could be returned, or the reference could be returned using a split-phase function invocation operation.
- In order to support separate compilation, how would the interface to such threaded functions be defined?

D. A THREADED function could be made to return a result using a split-phase function-call assignment operation. Figure 6 presents an example illustrating this new style of function call and return.

Rationale:

- This is a nice generalization of the idea of `THREADED` simply meaning "contains more than one fiber". The fact that a function has multiple fibers and that it can return a result should be orthogonal properties of the function. The `THREADED` keyword thus becomes like a storage class specifier such as `static` or `extern`. Also note that the return type information together with the `THREADED` keyword allow all appropriate typing information to be available when performing type-checking, even with separate compilation (use of header file).
- The introduction of the `RETURN` keyword is required to distinguish this function termination from a plain `TERMINATE` — the latter would be used only for function of type "`THREADED void`". Note that the fact that `RETURN` combines a result with a sync slot signal makes it similar to an ordinary `PUT_SYNC`.
- The use of a new operator ("`:=`") for performing a parallel call with a split-phase function return assignment is important to make it *explicit* that it is a parallel function that is being performed and that the variable to which the function result is being "assigned" cannot be used until the appropriate sync signal has been received.

  Using the regular "`=`" operator would be a bad use of overloading here, since the same symbol would be used for two *very* different operators: a regular assignment operator where the result is available right away after the instruction vs. a split-phase, non-immediate assignment, where the result is *not* be available after the instruction but only when some appropriate signal has been received.
- Note that this is a controversial feature. It will not be included in release 2.0 for lack of time *and* for lack of consensus.

E. If the `RETURN` instruction and split-phase assignment presented in the previous item were introduced in the language, a possible additional extension, that could be done without going to the full expressive power of fibers with formal arguments, could be to define fibers not in terms of the required number of sync signals they require, but instead in terms of the data on which they depend before they can proceed. An example is presented in Appendix C.

Acknowledgments

As mentioned in the introduction, this Technical Note is mostly a summary of the numerous discussions — through emails, in the hallways, at lunch, during Open Source Task Force meetings — that occurred during Winter and Spring 2000. These very interesting and fruitful discussions involved the following persons (listed in alphabetical order): Nelson J. Amaral, Mark D. Butala, Guang R. Gao, Christopher J. Morrone, Sean Ryan, and Kevin B. Theobald.

References

[1] José Nelson Amaral, Zachary Ruiz, Sean Ryan, Andres Marquez, Christopher Morrone, Prasad Kakulavarapu, and Guang R. Gao. Portable Threaded-C release 1.1. CAPSL Technical Note 05, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, October 1998.

[2] Kevin B. Theobald. Proposed changes to Threaded-C. CAPSL Technical Note 03, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, January 1998.

[3] Kevin B. Theobald. Definition of the EARTH model. CAPSL Technical Memo 33, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, October 1999. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[4] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

A Results from the (informal) survey on Threaded-C

As a prelude to this work, the author of this note, following a suggestion made by Dr. N. Amaral, made a small informal survey on Threaded-C, interviewing about 15 persons, mostly graduate students and post-docs having had some experience with Threaded-C programming.

Most of these people had previous programming experience mostly in C and, except for 3 persons (2 post-docs), none had any real previous experience in parallel programming. Note also that most people only had experience in Threaded-C developing small toy programs; only 3 had significant experience developing applications, although a few had experience *from the inside* (that is, they developed RTS-related stuff).

The key issues which were raised are presented below. Note that those annotated with (*) or (**) indicate issues which were mentioned more frequently than the others.

A. Things which make Threaded-C slightly difficult to learn and master:

   (*) `GLOBAL` handles. This is especially the case when there is multiple level of indirections, e.g., `int GLOBAL* GLOBAL* pt`.
   - Lot of special keywords.
   - `SYNC_SLOTS` are hard to manage. That is, it is not always clear how to correctly add the appropriate synchronization.

B. Features which are found particularly interesting or useful:

   (*) Use of `GLOBAL` handles for remote accesses.
   (*) Fibers and fine-grain parallelism.
   (*) Choice between `TOKEN` vs. `INVOKE`.
   - Detailed control over the asynchronous communications, for example, for consumer/producer communication. That is, programmer's control over how the fibers are connected together and how they synchronize.
   - Activation of threads through synchronization slots.

C. Features that are found awkward or annoying:

   (**) Poor support for debugging.
   (*) GLOBAL handles. Lack of pointer arithmetic for `GLOBAL` pointers, that is, they don't behave like normal pointers.

- Not obvious to set-up produced/consumer relationship. That is, numerous "tricks" have to be used to pass the correct addresses around.
- Handling of global variables.
- BLKMOV.
- Lot of keywords.
- Various forms of DATA/GET_SYNC_X instead of having a generic one.
- END_FUNCTION keyword, which must be the last instruction in the text of the function (static).
- Having to initialize the slots at the beginning of the THREADED function.

D. Features that should be improved, possible extensions:

(*) Better support for debugging.

(*) GLOBAL handles should be dropped or should be made first-class, e.g., dereference and global/local should be implicit, it should be possible to compute their size should, etc.

- Synchronization slots should be made into first class objects.
- Some form of parameterized thread should be allowed.
- Better integration of I-Structures (new data type?).
- An appropriate shared memory model should be defined (required when multiple threads execute concurrently on the same processor).
- A form of TOKEN whose migration can be under control of user should be supported, so the programmer has a better control over scheduling and threads.
- Should have some form of support for passing frame around.
- It should be possible to initialize a synchronization slot at the beginning of the corresponding thread.
- Should provide some form of support for data parallelism, parallel loop, etc.
- Should provide some support for hiding I/O latency, that is, support for parallel I/O.
- Should become an OO language.

Until the last person (Dr. Amaral) was interviewed, the author's own pet peeve about Threaded-C had not been mentioned by any of the interviewees: in Threaded-C as described in [4], threads are identified exclusively by numbers rather than by meaningful names. Furthermore, because of the existence of the implicit thread 0, thread 1 is usually associated with slot 0, thread 2 with slot 1, etc. This often leads to code which looks like the following, thus which is very difficult to understand:

```
INIT_SYNC( 0, 2, 2, 1 );
INIT_SYNC( 1, 3, 3, 2 );
INIT_SYNC( 2, 2, 2, 3 );
```

In the typical case where a slot is used for only one thread and vice-versa, it also leads to code where you have to SYNC(0) but you have to SPAWN(1).

Note that it was only after having interviewed Dr. Amaral that the author of this note learned that this last point had been fixed ... but not many persons seemed to use it or be aware of it. The use of symbolic slot and fiber names had been made possible in release 1.1 of Threaded-C [1], but this did not seem to have been made widely public.

```
THREADED fib( int n, ENTRY(int) result )
{
   if (n <= 1) {
      SEND( ARG1_READY, 0 );
      SEND( ARG2_READY, 1 );
   } else {
      TOKEN( fib, n-1, MK_ENTRY(ARG1_READY) );
      TOKEN( fib, n-2, MK_ENTRY(ARG2_READY) );
   }

   FIBER ARG1_READY (int r1)
         ARG2_READY (int r2) {
     SEND( result, r1+r2 );
     TERMINATE;
   }
}


THREADED MAIN( int argc, char *argv[] )
{
   int n;

   n = atoi(argv[1]);
   TOKEN( fib, n, MK_ENTRY(DONE) );

   FIBER DONE (int res) {
     printf( "fib(%d) = $d\n", n, res );
     TERMINATE;
   }
}
```

Figure 7: Fibers with arguments: `fib`

B Examples of Threaded-C programs using fibers with arguments

The examples presented in this section illustrate what Threaded-C programs might look like if fibers were allowed to have formal arguments, which might lead to a more explicit and clearer description of the flow of data and would preclude *pathological* coupling between fibers and threaded functions.

The first example (fig. 7) presents the classical Fibonacci program. Note that, in all this example and the followings, the type `ENTRY(t1, ..., tk)` denotes a fiber entry point receiving arguments of type `t1, ..., tk`. The second example (fig. 8) is a revised version of the N-queens example of [4], where only the `queens` procedure has been given (the `MAIN` is trivial).

The last example (fig. 9) illustrates the setting-up of a reduction process with multiple producers and a single consumer. Note that this example does use the `EXCLUSIVE` annotation; however, it does not use an atomic mailbox. For this example to work correctly, we would have to assume that the RTS would take care of buffering the arguments and would copy them locally into the frame only when the fiber's execution is to be started — of course, this also means that this program might not work correctly if

```
THREADED queens
   (
   int n, int row, int start_col, int *GLOBAL previous,
   ENTRY(int) result_ready
   )
{
  int col,
    own_board[24],
    sols_this_col,
    sols_other_cols;

  BLKMOV_SYNC( previous, TO_GLOBAL(own_board),
               row*sizeof(int), READY );

  FIBER READY <1> {
    for( col = start_col; col < n; col++ ) {
      if( safe(own_board, row, col) ) {
        own_board[row] = col;
        if (row+1 == n) {
          SEND( DONE1, 0 );
        } else {
          TOKEN( queens, n, row+1, 0, TO_GLOBAL(own_board),
                 MK_ENTRY(DONE1) );
        }
        if (col+1 == n) {
          SEND( DONE2, 0 );
        } else {
          TOKEN( queens, n, row, col+1, TO_GLOBAL(own_board),
                 MK_ENTRY(DONE2) );
        }
        break;
      }
    }
    if (col == n){
      SEND( DONE1, 0 );
      SEND( DONE2, 0 );
    }
  }

  FIBER DONE1 (int sols_this_col)
        DONE2 (int sols_other_cols) {
    SEND( result_ready, sols_this_col + sols_other_cols );
    TERMINATE;
  }
}
```

Figure 8: Fibers with arguments: `queens`

`EXCLUSIVE` was omitted.

```
typedef ENTRY(int, SPTR) ApplyFiber;
typedef ENTRY(int) RcvIntFiber;

THREADED perform_reduction
   (
   int init_value,        /* Initial value (neutral element). */
   int (*op)(int, int),   /* Binary operation to apply. */
   /* To signal when initialization has been done. */
   ENTRY(ApplyFiber, ENTRY(RcvIntFiber)) init_done
   )
{
  int current_value = init_value;

  SEND( init_done, MK_ENTRY(APPLY), MK_ENTRY(SEND_RESULT) );

  EXCLUSIVE FIBER APPLY( int new_value, SPTR done ) {
    current_value = (*op)(current_value, new_value);
    SYNC( done );
  }

  FIBER SEND_RESULT( RcvIntFiber rcv_result ) {
    SEND( rcv_result, current_value );
    TERMINATE;
  }
}


/* The various invocations of the following function generate the
 * values to be reduced and call the fiber that effectively performs
 * the reduction.  */

THREADED generate_value
   (
   int i,            /* Local identifier. */
   ApplyFiber apply, /* The fiber that performs the reduction. */
   SPTR done         /* To signal when the value has been reduced. */
   )
{
  v = someComputation(i);  /* Computation that generates a value. */
  SEND( apply, v, done );
  TERMINATE;
}
```

C Examples of Threaded-C programs using fibers with data dependencies

The `RETURN` instruction and split-phase assignment presented in Section 8 could be extended as presented in Figure 10, where fibers are defined not in terms of the required number of sync signals they need, but

```
/* The binary operation that will be used as reduction operation.  */
int max( int a1, int a2 ) { return( a1 < a2 ? a2 : a1 ); }


THREADED MAIN()
{
  ENTRY(RcvIntFiber) send_result;

  /* Initialize the reduction function. */
  TOKEN( perform_reduction, 0, max, MK_ENTRY(INIT_DONE) );

  /* Receive reduction fiber. */
  FIBER INIT_DONE( ApplyFiber _apply,
                   ENTRY(RcvIntFiber) _send_result ) {
    send_result = _send_result;
    for ( int i = 0; i < N; i++ ) {
      TOKEN( generate_value, i, apply, TO_SPTR(ALL_DONE) );
    }
  }

  FIBER ALL_DONE <N> {
    SEND( send_result, MK_ENTRY(PRINT_RESULT) );
  }

  FIBER PRINT_RESULT( int res ) {
    printf( "Result is %d\n", res );
    TERMINATE;
  }
}
```

Figure 9: Fibers with arguments: A reduction process with multiple producers

instead in terms of the data on which they depend. Suppose a fiber is defined as follows:

```
FIBER F <# x1, x2, ..., xk #> { ... }
```

Then such a fiber can proceed as soon as signals have been received from split-phase assignments to x1, . . . , xk.

Note that if it is assumed that such a variable can only appear in a single fiber *and* that a RETURN can only be used in the context of a split-phase assignment, and that all such split-phase assignments can only be used with such fibers, then this means the sync slot could also be omitted, which is what has been done in Figure 10.

```
#include <stdio.h>
#include <stdlib.h>

THREADED int fib( int n )
{
  int r1, r2;

  if (n <= 1) {
    r1 := 0;
    r2 := 1;
  } else {
    r1 := TOKEN( fib, n-1 );
    r2 := TOKEN( fib, n-2 );
  }

  FIBER READY <# r1, r2 #> {
    RETURN( r1+r2 );
  }
}

THREADED void MAIN( int argc, char* argv[] )
{
  int n, res;

  n = atoi(argv[1]);
  res := TOKEN( fib, n );

  FIBER FIB_DONE <# res #> {
    printf( "fib(%d) = %d\n", n, res );
    TERMINATE;
  }
}
```

Figure 10: A program with fibers and their associated data dependencies