```
#include <pthread.h>
#include <iostream>
#include <map>
#include <vector>
#include <string.h>
#include <signal.h>
#include <set>
using namespace std;

// Definition of type script_function, that masks a void* (*)(void*)
typedef void* (*script_function)(void*);
// Definition of type script_vector, a std::vector of type script_function
typedef std::vector<script_function> script_vector;

// Definition of type _single_data_readers_log, that masks a std::vector of type int

//
// structure: _single_data_readers_log, used to encapsulate a vector of type int
//            containing the positions of all the threads that have read a particular
//            data item, as well as a mutex for synchonized use.
//
struct _single_data_readers_log {

        vector <int> _readers;
        pthread_mutex_t _mutex;

        _single_data_readers_log(){
                pthread_mutex_init (&_mutex, NULL);
        };

};

//
// structure: _data_value_spec_thread, used to encapsulate a pthread,
//           which will run a speculative thread for data value speculation,
//           keeping as well some related data.
//
struct _data_value_spec_thread{

        pthread_t _thread; //thread and mutex to manage it
        pthread_mutex_t _thread_mutex;

        void* (*_thread_instructions) (void*); //thread instructions and arguments, necessary
        void* _const_args;                     //to create a thread again.

        //red-black tree for keeping the per-thread data copies.
        //These copies help to prevent antidependence violations and
        //output violations (WAR and WAW).
        map <void*, _data_copy> _copied_data;   //The _data_copy type is defined in branch_speculator.h

        //set for tracking the shared_data data that the thread has read.
        set <void*> _read_data;

        //
        //default constructor, initializes the mutexes
        //
        _data_value_spec_thread(){
                pthread_mutex_init (&_thread_mutex, NULL);
        };

};


//!
//! \class  data_value_speculator
//!
//! \brief  implements a class that takes n conditional branches and
//!         executes them speculatively, until one of them is
//!         proven valid in the pre-branch section. In order to do this
//!         the class relies on write_data and read_data functions, that
//!         check for data dependence violations; and 3 other functions
//!         (validate_supposition, speculate and get_results),
//!         that are in charge of control dependence.
//!         Additional functionality is provided with an append function
//!         that allows the pre-branch to dynamically add new branches
//!         to an on-going speculative execution.
//!
//! Limitations:
//! * If a speculative thread is to be canceled, it cannot use functions
//! that involve system mutexes, such as printf, etc. In this case, it
//! is possible that the thread can be canceled while holding such a mutex,
//! and the application con go into deadlock. In order to prevent this the
//! user has to surround this "dangerous" code with:
//!     "pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);" and
```

```
//!     "pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);".
//!
//! * Sequential consistency is mostly guaranteed, save for exception
//! behaviour.
//!

class data_value_speculator
{

private:

        //bool values to set if the speculator is active, or has control
        //of the pre-branch section
        bool _is_active, _has_pre_branch;

        //mutex for synchronisation of the previous group of variables
        pthread_mutex_t _is_active_mutex;

        //thread to manage the pre-branch & related mutex
        pthread_t _pb;
        pthread_mutex_t _pb_mutex;

        //the speculative threads and it's data
        vector <_data_value_spec_thread> _spec_threads;

        //red-black tree, used as a thread index to relate a pthread_id with
        //it's model id, i.e. it's position in _spec_threads. -1 is used to
        //indicate a thread in deferred cancel
        map <pthread_t, int> _thread_index;

        //mutex for a synchronized access to both of the former
        pthread_mutex_t _spec_threads_mutex;

        //shared_data data between the pre-branch & the speculative threads
        void*& _shared_data;

        //auxiliary data to reset _shared_data and prevent segmentation faults.
        int _null_data;

        //position of the valid thread, -1 if none has been validated.
        int _valid_thread;

        //mutex for a synchronized access to the former
        pthread_mutex_t _valid_thread_mutex;

        //red-black tree that relates the reference of a data element from
        //shared_data data with a vector of type int used to keep track of all the
        //readers of that particular data element.
        map <void*, _single_data_readers_log> _readers_log;

        //global mutex related to the _readers_log as a whole
        pthread_mutex_t _global_readers_log_mutex;


        //!
        //! private method to reset or cancel speculative threads
        //!
        //! TO BE NOTED:
        //! * this method takes all class mutexes. On invocation
        //!   _spec_threads_mutex and _global_readers_log_mutex should
        //!    be on hold and no other class mutex, save those related
        //!   to a branch that is not about to be canceled or restarted.
        //! * If given reset_all_logs=true && cancel=true and a vector
        //!   threads_to_delete of size 0, the object is initialized
        //!   only deleting the valid_thread from the previous run, if
        //!   it exists.
        //!
        void _reset_spec_threads (vector <int> threads_to_delete, bool reset_all_logs, bool cancel){

                int i;
                bool valid_arguments=true;

                if (!threads_to_delete.empty()){

                        for (i=0; i<threads_to_delete.size(); i++){

                                if (threads_to_delete[i]<0 || threads_to_delete[i]>_spec_threads.size()){
                                        valid_arguments=false; //one thread has an invalid id.
                                        i=threads_to_delete.size();
                                }

                        }
                }
                else{ //no threads to delete
```

```
        valid_arguments=false;

        /*This option will be used to initialize the object, it only cancels the valid thread from the previous run
        and resets the object's arrays*/

        if (reset_all_logs && cancel){  //all the readers log will be deleted

                pthread_mutex_lock(&_valid_thread_mutex);
                int valid_thread=_valid_thread;
                pthread_mutex_unlock(&_valid_thread_mutex);

                if (valid_thread>0 && valid_thread<_spec_threads.size())

                        pthread_mutex_lock(&_spec_threads[valid_thread]._thread_mutex);

                pthread_mutex_lock(&_is_active_mutex);

                pthread_mutex_lock(&_valid_thread_mutex);

                if (_has_pre_branch)

                        pthread_mutex_lock(&_pb_mutex);

                if (!_spec_threads.empty()){
                        if (valid_thread>0 && valid_thread<_spec_threads.size()){
                                if (pthread_kill(_spec_threads[valid_thread]._thread, 0)==0)
                                        pthread_cancel(_spec_threads[valid_thread]._thread);
                                _thread_index[_spec_threads[valid_thread]._thread]= -1;
                                pthread_mutex_destroy(&_spec_threads[valid_thread]._thread_mutex);
                        }
                        _spec_threads.clear();
                }

                if (!_readers_log.empty()){
                        _readers_log.clear();
                }

                if (!_thread_index.empty()){

                /*in order to clear the _thread_index, it will be necessary to guarantee that all the threads in deferred
                cancel have finished, and thus will not use the object*/

                        if (_has_pre_branch)

                                pthread_mutex_unlock(&_pb_mutex);

                        pthread_mutex_unlock(&_valid_thread_mutex);

                        pthread_mutex_unlock(&_is_active_mutex);

                        pthread_mutex_unlock(&_spec_threads_mutex);

                        pthread_mutex_unlock(&_global_readers_log_mutex);

                        map <pthread_t, int>::iterator it;
                        for (it=_thread_index.begin(); it!=_thread_index.end(); it++){
                                if (it->second==-1){
                                        int a;
                                        do {
                                                a= pthread_kill(it->first, 0);
                                        } while (a==0);
                                }
                        }

                        pthread_mutex_lock(&_global_readers_log_mutex);

                        pthread_mutex_lock(&_spec_threads_mutex);

                        _thread_index.clear();
                }
                else {

                        if (_has_pre_branch)

                                pthread_mutex_unlock(&_pb_mutex);

                        pthread_mutex_unlock(&_valid_thread_mutex);

                        pthread_mutex_unlock(&_is_active_mutex);

                }

        }
}

if (valid_arguments) {
```

```
for (i=threads_to_delete.size()-1; i>=0; i--){

        pthread_mutex_lock(&_spec_threads[threads_to_delete[i]]._thread_mutex);
}


if (reset_all_logs && cancel){  //all the readers log will be deleted, as well as all the threads


        pthread_mutex_lock(&_is_active_mutex);

        pthread_mutex_lock(&_valid_thread_mutex);

        if (_has_pre_branch)

                pthread_mutex_lock(&_pb_mutex);

        for (i=0; i<threads_to_delete.size(); i++){

                if (pthread_kill(_spec_threads[threads_to_delete[i]]._thread, 0)==0)

                        pthread_cancel(_spec_threads[threads_to_delete[i]]._thread);

                _thread_index[_spec_threads[threads_to_delete[i]]._thread]= -1;

                pthread_mutex_destroy(&_spec_threads[threads_to_delete[i]]._thread_mutex);
        }


        if (_has_pre_branch)
                pthread_mutex_unlock(&_pb_mutex);

        pthread_mutex_unlock(&_valid_thread_mutex);

        pthread_mutex_unlock(&_is_active_mutex);

}
else {

        //next it will be checked what data has been read by the threads_to_delete.

        set <void*> data_to_access;

        for (i=threads_to_delete.size()-1; i>=0; i--){

                if (!_spec_threads[threads_to_delete[i]]._read_data.empty()){

                        set <void*>::iterator it;

                        for (it=_spec_threads[threads_to_delete[i]]._read_data.begin(); it!=_spec_threads[threads_
                                if (data_to_access.find(*it)==data_to_access.end()){

                                        data_to_access.insert(*it);
                                }
                        }

                        _spec_threads[threads_to_delete[i]]._read_data.clear();

                        _spec_threads[threads_to_delete[i]]._copied_data.clear();
                }
        }

        if (data_to_access.empty()){

                pthread_mutex_lock(&_is_active_mutex);

                pthread_mutex_lock(&_valid_thread_mutex);

                if (_has_pre_branch)

                        pthread_mutex_lock(&_pb_mutex);

                for (i=0; i<threads_to_delete.size(); i++){

                        if (pthread_kill(_spec_threads[threads_to_delete[i]]._thread, 0)==0){
                                pthread_cancel(_spec_threads[threads_to_delete[i]]._thread);

                        }
                        _thread_index[_spec_threads[threads_to_delete[i]]._thread]= -1;

                }

                if (!cancel) { //this means that the threads will be restarted

                        pthread_attr_t attr;
                        pthread_attr_init (&attr);
```

```
                pthread_attr_setschedpolicy(&attr, SCHED_RR);

                for (i=0; i<threads_to_delete.size(); i++){
                        int success=0;
                        do{
                        success=pthread_create(&_spec_threads[threads_to_delete[i]]._thread, &attr, _spec_
                        } while (success!=0);
                        _thread_index[_spec_threads[threads_to_delete[i]]._thread]= threads_to_delete[i];
                }
        }

        if (_has_pre_branch)

                pthread_mutex_unlock(&_pb_mutex);

        pthread_mutex_unlock(&_valid_thread_mutex);

        pthread_mutex_unlock(&_is_active_mutex);
}
else{
        pthread_mutex_lock(&_is_active_mutex);

        pthread_mutex_lock(&_valid_thread_mutex);

        if (_has_pre_branch)

                pthread_mutex_lock(&_pb_mutex);

        for (i=0; i<threads_to_delete.size(); i++){

                if (pthread_kill(_spec_threads[threads_to_delete[i]]._thread, 0)==0){
                        pthread_cancel(_spec_threads[threads_to_delete[i]]._thread);

                }

                _thread_index[_spec_threads[threads_to_delete[i]]._thread]= -1;
        }

        if (reset_all_logs){

                set <void*>::iterator it;

                for (it=data_to_access.begin(); it!=data_to_access.end(); it++){

                        if (_readers_log.find(*it)!=_readers_log.end()){


                                _single_data_readers_log * ptr= &_readers_log.find(*it)->second;


                                pthread_mutex_lock (&ptr->_mutex);

                                if (!ptr->_readers.empty()){

                                        ptr->_readers.clear();
                                }

                                pthread_mutex_unlock (&ptr->_mutex);


                        }
                }
        }
        else{ //only the threads to delete will be deleted from the logs

                set <void*>::iterator it;

                for (it=data_to_access.begin(); it!=data_to_access.end(); it++){


                        if (_readers_log.find(*it)!=_readers_log.end()){

                                _single_data_readers_log * ptr= &_readers_log.find(*it)->second;


                                pthread_mutex_lock (&ptr->_mutex);


                                if (!ptr->_readers.empty()){

                                        for (i=0; i<threads_to_delete.size(); i++){

                                                vector<int> toErase;

                                                for (int j=0; j<ptr->_readers.size(); j++){
                                                        if (ptr->_readers[j]==threads_to_delete[i]){
                                                        }
                                                }
```

```
                                                                        if (!toErase.empty()){
                                                                                int subs=0;
                                                                                for (int k=0; k<toErase.size(); k++){
                                                                                        ptr->_readers.erase(ptr->_readers.begin()+
                                                                                        subs++;
                                                                                }
                                                                        }
                                                                }
                                                        }

                                                        pthread_mutex_unlock (&ptr->_mutex);
                                                }
                                        }
                                }

                                if (!cancel) { //this means that the threads will be restarted
                                        pthread_attr_t attr;
                                        pthread_attr_init (&attr);
                                        pthread_attr_setschedpolicy(&attr, SCHED_RR);

                                        for (i=0; i<threads_to_delete.size(); i++){
                                                int success=0;
                                                do {success=pthread_create(&_spec_threads[threads_to_delete[i]]._thread, &attr, _s
                                                } while (success!=0);
                                                _thread_index[_spec_threads[threads_to_delete[i]]._thread]=threads_to_delete[i];
                                        }
                                }

                                if (_has_pre_branch)

                                        pthread_mutex_unlock(&_pb_mutex);

                                pthread_mutex_unlock(&_valid_thread_mutex);

                                pthread_mutex_unlock(&_is_active_mutex);

                        }
                }
        }
        if ( valid_arguments && !(reset_all_logs&&cancel)){

                for (i=0; i<threads_to_delete.size(); i++){
                        pthread_mutex_unlock(&_spec_threads[threads_to_delete[i]]._thread_mutex);
                }
        }
};


public:


        //!
        //! default constructor
        //!
        data_value_speculator():_shared_data((void*&)_null_data){
                _is_active=false;
                pthread_mutex_init (&_is_active_mutex, NULL);
                pthread_mutex_init (&_pb_mutex, NULL);
                pthread_mutex_init (&_spec_threads_mutex, NULL);
                pthread_mutex_init (&_valid_thread_mutex, NULL);
                pthread_mutex_init (&_global_readers_log_mutex, NULL);
                _valid_thread=-1;//no branch has been validated
                _null_data=-1;
        };


        //!
        //! function providing access to the shared_data as a whole
        //!
        void*& get_shared_data (){

                pthread_mutex_lock(&_is_active_mutex);

                bool manages_pre_branch=_has_pre_branch;
                if (!_is_active){

                        pthread_mutex_unlock(&_is_active_mutex);

                        return (void*&)_null_data; //the object is inactive

                }

                pthread_mutex_unlock(&_is_active_mutex);

                pthread_mutex_lock(&_spec_threads_mutex);

                if (_thread_index.empty() ||_thread_index.find(pthread_self())!=_thread_index.end()){
```

```
                pthread_mutex_unlock(&_spec_threads_mutex);

                return _shared_data; //valid branch or branch in deferred cancelation
        }
        pthread_mutex_unlock(&_spec_threads_mutex);

        if (manages_pre_branch){

                pthread_mutex_lock(&_pb_mutex);

                if (pthread_self()==_pb){

                        pthread_mutex_unlock(&_pb_mutex);
                        return _shared_data;
                }

                else {
                        pthread_mutex_unlock(&_pb_mutex);

                        return (void*&)_null_data; //invalid caller
                }
        }


        return _shared_data; //unmanaged pre-branch or branch in deferred cancelation
};


//!
//! function allowing to validate one of the threads of the model
//! according to it's position
//!
//! TO BE NOTED:
//! * has to be called from the pre-branch section
//!
int validate_supposition (int thread_to_validate){
        pthread_mutex_lock(&_is_active_mutex);

        if (!_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1; //the object is inactive
        }

        bool manages_pre_branch=_has_pre_branch;

        pthread_mutex_unlock(&_is_active_mutex);

        bool is_pre_branch=false;

        pthread_mutex_lock(&_valid_thread_mutex);
        if (_valid_thread!=-1){
                pthread_mutex_unlock(&_valid_thread_mutex);
                return -1;
        }
        pthread_mutex_unlock(&_valid_thread_mutex);


        if (manages_pre_branch){

                pthread_mutex_lock(&_pb_mutex);

                if (pthread_self()!=_pb){
                        pthread_mutex_unlock(&_pb_mutex);
                        return -1; //invalid caller
                }
                pthread_mutex_unlock(&_pb_mutex);
                is_pre_branch=true;
        }


        pthread_mutex_lock(&_spec_threads_mutex);

        if (!is_pre_branch){

                if (thread_to_validate<0){
                                pthread_mutex_unlock(&_spec_threads_mutex);
                                return -1; //invalid argument
                }
                if (_thread_index.find(pthread_self())!=_thread_index.end()){
                                pthread_mutex_unlock(&_spec_threads_mutex);
                                return -1;//invalid caller
                }
        }


        pthread_mutex_lock(&_valid_thread_mutex);
```

```
        _valid_thread=thread_to_validate;
        pthread_mutex_unlock(&_valid_thread_mutex);

        if (thread_to_validate<_spec_threads.size()){
                vector <int> threads_to_delete;
                for (int i=0; i<_spec_threads.size(); i++){
                        if (i!=thread_to_validate)
                                threads_to_delete.push_back(i);
                }
                if (!threads_to_delete.empty()){
                        //the threads that were invalidated are deleted
                        _reset_spec_threads(threads_to_delete, false, true);
                }
        }
        pthread_mutex_unlock(&_spec_threads_mutex);
        return 0;

};

//!
//! function to be called from the speculative branches
//! and pre-branch, in order to read the shared_data
//! while keeping the expected sequential data consistency
//!
template <class T>
T read_data(T*& data_to_be_read){


        pthread_mutex_lock(&_is_active_mutex);
        if (!_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return *(T*)data_to_be_read; //the object is inactive, it's calling data is returned instead
                                             //of NULL, to prevent segmentation faults.
        }
        bool manages_pre_branch=_has_pre_branch;
        pthread_mutex_unlock(&_is_active_mutex);

        T retval;
        bool thrd_id_equals_pb=false;


        if (manages_pre_branch){

                pthread_mutex_lock(&_pb_mutex);

                thrd_id_equals_pb=(pthread_self()==_pb);

                pthread_mutex_unlock(&_pb_mutex);

        }

        if (thrd_id_equals_pb){
                retval= *(T*) data_to_be_read;
                return retval;//the pre-branch does a standard read
        }

        pthread_mutex_lock(&_spec_threads_mutex);

        pthread_t thrd_id=pthread_self();

        //the branches need to make a copy or read an already existing copy, and log the reading.

        int current_thread=-1;

        if (_thread_index.empty()){
                        retval= *(T*) data_to_be_read;
                        pthread_mutex_unlock(&_spec_threads_mutex);
                        return retval;

        }
        if (_thread_index.find(thrd_id)!=_thread_index.end()){
                current_thread=_thread_index.find(thrd_id)->second;
        }
        if (current_thread>=0){

                _data_value_spec_thread* thrd_ptr=&_spec_threads[current_thread];

                pthread_mutex_lock(&thrd_ptr->_thread_mutex);

                map <void*, _data_copy>::iterator it;

                it=thrd_ptr->_copied_data.find((void*)data_to_be_read);

                if (it!=thrd_ptr->_copied_data.end()){
                                retval=(T)it->second._data;
                                pthread_mutex_unlock(&thrd_ptr->_thread_mutex);
                                pthread_mutex_unlock(&_spec_threads_mutex);
```

```
                                return retval;//if a copy is found, the copy is read
                        }

                        if (thrd_ptr->_read_data.find((void*)data_to_be_read)!=thrd_ptr->_read_data.end()){
                                retval= *(T*) data_to_be_read;
                                pthread_mutex_unlock(&thrd_ptr->_thread_mutex);

                                pthread_mutex_unlock(&_spec_threads_mutex);

                                return retval;//if the data has already been read, then the reading neet not be logged.
                        }

                        thrd_ptr->_read_data.insert((void*)data_to_be_read);//the read is logged in the thread

                        pthread_mutex_lock (&_global_readers_log_mutex);

                        if (_readers_log.find((void*)data_to_be_read)!=_readers_log.end()){

                                _single_data_readers_log * ptr= &_readers_log.find((void*)data_to_be_read)->second;

                                pthread_mutex_lock(&ptr->_mutex);

                                pthread_mutex_unlock(&_global_readers_log_mutex);

                                ptr->_readers.push_back(current_thread);//the read is marked in the data

                                retval= *(T*) data_to_be_read;

                                pthread_mutex_unlock(&ptr->_mutex);

                                pthread_mutex_unlock(&thrd_ptr->_thread_mutex);

                                pthread_mutex_unlock(&_spec_threads_mutex);

                                return retval;
                        }
                        else{// the data log has to be created
                                pthread_mutex_t temp;
                                pthread_mutex_init (&temp, NULL);
                                _single_data_readers_log temp_log;
                                temp_log._readers.push_back(current_thread);
                                _readers_log.insert(pair<void*, _single_data_readers_log>((void*)data_to_be_read, temp_log));
                                retval= *(T*) data_to_be_read;

                                pthread_mutex_unlock(&_global_readers_log_mutex);

                                pthread_mutex_unlock(&thrd_ptr->_thread_mutex);

                                pthread_mutex_unlock(&_spec_threads_mutex);

                                return retval;
                        }
                }
                if (!manages_pre_branch){
                        retval=*(T*) data_to_be_read;
                        pthread_mutex_unlock(&_spec_threads_mutex);
                        return retval;//un-managed pre-branch
                }

                pthread_mutex_unlock(&_spec_threads_mutex);
                return *(T*)data_to_be_read; //invalid caller or branch in deferred cancelation.
                                             //it's calling data is returned instead of NULL, to prevent segmentation faults.
};


//!
//! function to be called from the speculative branches
//! and pre-branch, in order to write the shared_data
//! while keeping the expected sequential data consistency
//!
template <class T>
int write_data(T*& data_to_be_written_upon, T* data_to_write){

        pthread_mutex_lock(&_is_active_mutex);
        if (!_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1; //the object is inactive.
        }
        bool manages_pre_branch=_has_pre_branch;
        pthread_mutex_unlock(&_is_active_mutex);

        pthread_mutex_lock(&_valid_thread_mutex);
        int valid_thread=_valid_thread;
        pthread_mutex_unlock(&_valid_thread_mutex);

        bool thrd_id_equals_pb=false;
```

```
if (manages_pre_branch){
        pthread_mutex_lock(&_pb_mutex);
        thrd_id_equals_pb=(pthread_self()==_pb);
        pthread_mutex_unlock(&_pb_mutex);
}


pthread_mutex_lock(&_spec_threads_mutex);

pthread_t thrd_id=pthread_self();

bool is_pre_branch=false;
if (thrd_id_equals_pb){
        is_pre_branch=true;
}

int current_thread=-2;

if (_thread_index.empty()){
        pthread_mutex_unlock(&_spec_threads_mutex);
        return -1;
}

if (_thread_index.find(thrd_id)!=_thread_index.end()){
        current_thread=_thread_index.find(thrd_id)->second;
}

if (current_thread==-2 && !manages_pre_branch){
        is_pre_branch=true;
}

if (is_pre_branch){

        pthread_mutex_lock (&_global_readers_log_mutex);

        if (_readers_log.find((void*)data_to_be_written_upon)==_readers_log.end()){

                pthread_mutex_unlock(&_spec_threads_mutex);

                memcpy ((void*)data_to_be_written_upon, (void*)&data_to_write, sizeof(T));

                pthread_mutex_unlock(&_global_readers_log_mutex);

                return 0;//if no log exists for the data, then it is simply written.
        }

        //Otherwise the data is written, then data dependency checks are preformed, in search of a true
        //dependence violation (RAW).

        memcpy ((void*)data_to_be_written_upon, (void*)&data_to_write, sizeof(T));

        _single_data_readers_log *ptr=&_readers_log.find((void*)data_to_be_written_upon)->second;

        pthread_mutex_lock(&ptr->_mutex);

        pthread_mutex_unlock (&_global_readers_log_mutex);

        vector <int> delinquent_readers= ptr->_readers;

        pthread_mutex_unlock(&ptr->_mutex);

        if (!delinquent_readers.empty()){//a true data dependency violation has ocurred (RAW).

                _reset_spec_threads(delinquent_readers, false, false);
        }
        pthread_mutex_unlock(&_spec_threads_mutex);

        return 0;
}
if (current_thread>=0){

        _data_value_spec_thread* thrd_ptr= &_spec_threads[current_thread];
        pthread_mutex_lock(&thrd_ptr->_thread_mutex);
        pthread_mutex_unlock(&_spec_threads_mutex);

        //if the thread has a copy, it is written
        if (!thrd_ptr->_copied_data.empty()){
                if(thrd_ptr->_copied_data.find((void*)data_to_be_written_upon)!=thrd_ptr->_copied_data.end()){
                        thrd_ptr->_copied_data[(void*)data_to_be_written_upon]._data=(void*)data_to_write;
                        pthread_mutex_unlock(&thrd_ptr->_thread_mutex);
                        return 0;
                }
        }
        //if there is no per-thread copy, it is made and written.
        _data_copy copy;
        copy._data=(void*)data_to_write;
```

```
                copy._size=sizeof(T);
                thrd_ptr->_copied_data.insert(pair<void*, _data_copy>((void*)data_to_be_written_upon, copy));
                pthread_mutex_unlock(&thrd_ptr->_thread_mutex);

                return 0;
        }
        pthread_mutex_unlock(&_spec_threads_mutex);
        return -1;//invalid caller or branch in deferred cancelation.
};

//!
//! function that permits the pre_branch to dynamically append
//! a new speculative thread at the end of the array in an on-going
//! speculative execution.
//!
int append(void* (f)(void*), void* consts){

        pthread_mutex_lock(&_is_active_mutex);
        if (!_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1; //the object is not active.
        }
        bool manages_pre_branch=_has_pre_branch;
        pthread_mutex_unlock(&_is_active_mutex);

        bool is_pre_branch=false;
        if (manages_pre_branch){
                pthread_mutex_lock(&_pb_mutex);
                if (pthread_self()==_pb){
                        is_pre_branch=true;
                }
                pthread_mutex_unlock(&_pb_mutex);
        }

        int current_thread=-2;

        pthread_mutex_lock(&_spec_threads_mutex);
        pthread_t thrd_id=pthread_self();
        if (_thread_index.find(thrd_id)!=_thread_index.end()){
                current_thread=_thread_index.find(thrd_id)->second;
        }

        if (current_thread==-2 && !manages_pre_branch){
                is_pre_branch=true;
        }
        if (is_pre_branch){
                pthread_attr_t attr;
                pthread_attr_init (&attr);
                pthread_attr_setschedpolicy(&attr, SCHED_RR);
                int success=0;
                _spec_threads.resize(_spec_threads.size()+1);
                pthread_mutex_lock(&_spec_threads[_spec_threads.size()-1]._thread_mutex);
                pthread_mutex_lock(&_is_active_mutex);
                pthread_mutex_lock(&_valid_thread_mutex);
                if (manages_pre_branch)
                        pthread_mutex_lock(&_pb_mutex);
                do {
                success=pthread_create(&_spec_threads[_spec_threads.size()-1]._thread, &attr, f, consts);
                } while (success!=0);
                _thread_index[_spec_threads[_spec_threads.size()-1]._thread]=_spec_threads.size()-1;
                if (manages_pre_branch)
                        pthread_mutex_unlock(&_pb_mutex);
                pthread_mutex_unlock(&_valid_thread_mutex);
                pthread_mutex_unlock(&_is_active_mutex);
                pthread_mutex_unlock(&_spec_threads[_spec_threads.size()-1]._thread_mutex);
                pthread_mutex_unlock(&_spec_threads_mutex);

                return 0;
        }
        pthread_mutex_unlock(&_spec_threads_mutex);

        return -1;//invalid caller.
};


//!
//! speculate: a complete function that takes the instructions and arguments
//! of the branches and pre-branch; and starts their speculatively parallel
//! execution, while maintaining the sequential consistency. This function
//! returns in the &shared_data, the results of the computation.
//!
//! TO BE NOTED:
//!    * In this version of the function, the object is in control of the pre-
//!    branch. This means that on invocation, the caller blocks until the pre-
//!    branch and validated thread complete their execution.
//!
```

```
int speculate (void*& shared_data, void* (fpb)(void*), void* const_args_pb, script_vector thread_instructions, vector <void*> cons

        pthread_mutex_lock(&_is_active_mutex);
        if (_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1;//the object is active, hence no other speculation can be started.
        }
        if (thread_instructions.size()!=const_args_spec_threads.size()){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1;//invalid arguments
        }
        _is_active=true;
        pthread_mutex_unlock(&_is_active_mutex);

        pthread_mutex_lock(&_spec_threads_mutex);
        pthread_mutex_lock(&_global_readers_log_mutex);

        vector <int> threads_to_cancel;//a selective reset is performed, canceling only
        _reset_spec_threads(threads_to_cancel, true, true);//the valid thread from the previous run, if any.

        pthread_mutex_lock(&_is_active_mutex);
        _has_pre_branch=true;
        pthread_mutex_unlock(&_is_active_mutex);

        pthread_mutex_lock(&_valid_thread_mutex);
        _valid_thread=-1;
        pthread_mutex_unlock(&_valid_thread_mutex);

        _shared_data=shared_data;

        pthread_attr_t attr;
        pthread_attr_init (&attr);
        pthread_attr_setschedpolicy(&attr, SCHED_RR);

        int success=0;

        pthread_mutex_lock(&_pb_mutex);
        success=pthread_create (&_pb, &attr, fpb, const_args_pb);
        pthread_mutex_unlock(&_pb_mutex);

        if (success!=0){


                        pthread_mutex_unlock(&_global_readers_log_mutex);
                        pthread_mutex_unlock(&_spec_threads_mutex);

                        pthread_mutex_lock(&_is_active_mutex);
                        _is_active=false;
                        pthread_mutex_unlock(&_is_active_mutex);

                        return -1;//the pre-branch could not be created.
        }

        _spec_threads.resize(thread_instructions.size());

        for (int i=thread_instructions.size()-1; i>=0; i--){
                pthread_mutex_lock(&_spec_threads[i]._thread_mutex);
        }
        for (unsigned int i=0; i<thread_instructions.size(); i++){
                _spec_threads[i]._thread_instructions=thread_instructions[i];
                _spec_threads[i]._const_args=const_args_spec_threads[i];
                success=pthread_create (&_spec_threads[i]._thread, &attr,
                                        _spec_threads[i]._thread_instructions,
                                        _spec_threads[i]._const_args);
                _thread_index[_spec_threads[i]._thread]= i;
                pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);
                if (success!=0){
                        for (int j=i; j<thread_instructions.size(); j++){
                                pthread_mutex_unlock(&_spec_threads[j]._thread_mutex);
                        }
                        vector <int> threads_to_delete;
                        for (int j=0; j>i; j++){
                                threads_to_delete.push_back(j);
                        }
                        _reset_spec_threads(threads_to_delete, true, true);
                        pthread_mutex_unlock(&_global_readers_log_mutex);
                        pthread_mutex_unlock(&_spec_threads_mutex);

                        pthread_mutex_lock(&_is_active_mutex);
                        _is_active=false;
                        pthread_mutex_unlock(&_is_active_mutex);

                        return -1;//one thread could not be created.
                }
        }

        pthread_mutex_unlock(&_global_readers_log_mutex);
```

```
	pthread_mutex_unlock(&_spec_threads_mutex);

	int valid_thread;
	int joins_but_no_validation=0;

	do{

		success=pthread_join (_pb, NULL);

		if (success!=0){
			pthread_mutex_lock(&_valid_thread_mutex);
			if (_valid_thread==-1){
				success=0;
				joins_but_no_validation++;
			}
			else{
				valid_thread=_valid_thread;
			}
			if (joins_but_no_validation>10){

			/*just in case the validated supposition is not logged in a timely fasion, it is tried
			10 times to see if it get's validated, if not then the object asumes there was no validated
			branch.
			Although slightly un-elegant, the validity of this solution has not been disproved by empirical
			tests.*/

				success=-1;
				valid_thread=-1;
			}
			pthread_mutex_unlock(&_valid_thread_mutex);
		}


	} while (success==0);


	pthread_mutex_lock(&_spec_threads_mutex);

	if (valid_thread==-1)
		valid_thread=_spec_threads.size()+1;

	if (valid_thread>=_spec_threads.size()){
		vector <int> threads_to_delete;
		for (int i=0; i<_spec_threads.size(); i++){
			threads_to_delete.push_back(i);
		}
		_reset_spec_threads(threads_to_delete, true, true);
		pthread_mutex_unlock(&_spec_threads_mutex);

		pthread_mutex_lock(&_is_active_mutex);
		_is_active=false;
		pthread_mutex_unlock(&_is_active_mutex);

		return -1;//no speculative thread was validated, thus all are canceled save the pre-branch.
	}

	pthread_mutex_unlock(&_spec_threads_mutex);

	do{
		success=pthread_join (_spec_threads[valid_thread]._thread, NULL);
	}while (success==0);

	pthread_mutex_lock(&_spec_threads_mutex);
	pthread_mutex_lock(&_spec_threads[valid_thread]._thread_mutex);

	if (!_spec_threads[valid_thread]._copied_data.empty()){
		//the changes that the valid thread made are communicated unto the &_shared_data
		map <void*, _data_copy>::iterator i;
		for (i=_spec_threads[valid_thread]._copied_data.begin(); i!=_spec_threads[valid_thread]._copied_data.end(); i++){
			memcpy((void*&)const_cast<void*&>(i->first), (void*)&(i->second._data), i->second._size);
		}
		_spec_threads[valid_thread]._copied_data.clear();
	}
	pthread_mutex_unlock(&_spec_threads[valid_thread]._thread_mutex);

	pthread_mutex_unlock(&_spec_threads_mutex);

	pthread_mutex_lock(&_is_active_mutex);
	_is_active=false;
	pthread_mutex_unlock(&_is_active_mutex);

	return 0;

};
```

```
//!
//! speculate: a function that takes the instructions and arguments
//! of the branches , and starts their speculatively parallel execution,
//! while maintaining the sequential consistency with an unmanaged pre-branch.
//!
//! TO BE NOTED:
//!    * In this version of the function, the object is not in control of the pre-
//!    branch. This means that on invocation, the caller only blocks for the creation
//!    of the branches, and can resume it's execution as a possible pre-branch, until
//!    calling get_results(), when the caller blocks until the valid branch returns.
//!
int speculate (void*& shared_data, script_vector thread_instructions, vector <void*> const_args_spec_threads){
        pthread_mutex_lock(&_is_active_mutex);
        if (_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1;//the object is active, hence no other speculation can be started.
        }
        if (thread_instructions.size()!=const_args_spec_threads.size()){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1;//invalid arguments
        }
        _is_active=true;
        pthread_mutex_unlock(&_is_active_mutex);

        pthread_mutex_lock(&_spec_threads_mutex);
        pthread_mutex_lock(&_global_readers_log_mutex);

        vector <int> threads_to_cancel;//a selective reset is performed, canceling only
        _reset_spec_threads(threads_to_cancel, true, true);//the valid thread from the previous run, if any.

        pthread_mutex_lock(&_is_active_mutex);
        _has_pre_branch=false;
        pthread_mutex_unlock(&_is_active_mutex);


        pthread_mutex_lock(&_valid_thread_mutex);
        _valid_thread=-1;
        pthread_mutex_unlock(&_valid_thread_mutex);

        _shared_data=shared_data;

        pthread_attr_t attr;
        pthread_attr_init (&attr);
        pthread_attr_setschedpolicy(&attr, SCHED_RR);


        int success=0;
        _spec_threads.resize(thread_instructions.size());
        for (int i=thread_instructions.size()-1; i>=0; i--){
                pthread_mutex_lock(&_spec_threads[i]._thread_mutex);
        }
        for (int i=0; i<thread_instructions.size(); i++){
                _spec_threads[i]._thread_instructions=thread_instructions[i];
                _spec_threads[i]._const_args=const_args_spec_threads[i];
                success=pthread_create (&_spec_threads[i]._thread, &attr,
                                        _spec_threads[i]._thread_instructions,
                                        _spec_threads[i]._const_args);
                _thread_index[_spec_threads[i]._thread]= i;
                pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);
                if (success!=0){
                        for (int j=i; j<thread_instructions.size(); j++){
                                pthread_mutex_unlock(&_spec_threads[j]._thread_mutex);
                        }
                        vector <int> threads_to_delete;
                        for (int j=0; j>i; j++){
                                threads_to_delete.push_back(j);
                        }
                        _reset_spec_threads(threads_to_delete, true, true);
                        pthread_mutex_unlock(&_global_readers_log_mutex);
                        pthread_mutex_unlock(&_spec_threads_mutex);

                        pthread_mutex_lock(&_is_active_mutex);
                        _is_active=false;
                        pthread_mutex_unlock(&_is_active_mutex);
                        return -1;//a thread could not be created.
                }

        }
        pthread_mutex_unlock(&_global_readers_log_mutex);
        pthread_mutex_unlock(&_spec_threads_mutex);
        return 0;
};

//!
//! function to get the results of the speculation once the un-managed pre-
//! branch has ended it's execution.
```

```
	//!
	//! TO BE NOTED:
	//!    * If the pre-branch does not validate the supposition, then none of
	//!    the results of the branches will be accepted.
	//!
	int get_results(){
		pthread_mutex_lock(&_is_active_mutex);
		if (!_is_active || _has_pre_branch){
			pthread_mutex_unlock(&_is_active_mutex);
			return -1;//the object is active or manages it's pre-branch
		}
		pthread_mutex_unlock(&_is_active_mutex);


		pthread_mutex_lock(&_valid_thread_mutex);
		int valid_thread=_valid_thread;
		pthread_mutex_unlock(&_valid_thread_mutex);

		if (valid_thread==-1){
			pthread_mutex_lock(&_spec_threads_mutex);
			vector <int> threads_to_delete;
			for (int i=0; i<_spec_threads.size(); i++){
				threads_to_delete.push_back(i);
			}
			_reset_spec_threads(threads_to_delete, true, true);
			pthread_mutex_unlock(&_spec_threads_mutex);

			pthread_mutex_lock(&_is_active_mutex);
			_is_active=false;
			pthread_mutex_unlock(&_is_active_mutex);
			return -1; //no speculative thread was validated.
		}
		int a=0;

		do{
			a=pthread_join (_spec_threads[valid_thread]._thread, NULL);
		} while (a==0);

		pthread_mutex_lock(&_spec_threads_mutex);
		pthread_mutex_lock(&_spec_threads[valid_thread]._thread_mutex);
		if (!_spec_threads[valid_thread]._copied_data.empty()){
			//the changes that the valid thread made are communicated unto the &_shared_data
			map <void*, _data_copy>::iterator i;
			for (i=_spec_threads[valid_thread]._copied_data.begin(); i!=_spec_threads[valid_thread]._copied_data.end(); i++){
				memcpy((void*&)const_cast<void*&>(i->first), (void*)&(i->second._data), i->second._size);
			}
			_spec_threads[valid_thread]._copied_data.clear();
		}

		pthread_mutex_unlock(&_spec_threads[valid_thread]._thread_mutex);

		pthread_mutex_unlock(&_spec_threads_mutex);

		pthread_mutex_lock(&_is_active_mutex);
		_is_active=false;
		pthread_mutex_unlock(&_is_active_mutex);

		return 0;
	};

};
```