```
#include <pthread.h>
#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;



//
// structure: _loop_spec_thread, used to encapsulate a pthread,
//           which will run a speculative thread for loop speculation,
//           keeping as well some related data.
//
struct _loop_spec_thread{
        pthread_mutex_t _thread_mutex;
        pthread_t _thread;

        set <void*> _written_data; //will be used on thread-cancelation,
        //to undo any changes it has made.

        set <void*> _read_data; //will be used on threcad-cancelation,
        //to delete it's logs as reader.

        void* (*_thread_instructions) (void*);
        void* _const_args;

        bool _commit;//helps to determine if the loop iteration
        //that the _loop_spec_thread represents has commited or not.

        _loop_spec_thread(){
                pthread_mutex_init (&_thread_mutex, NULL);
                _commit=false;
        };

};
//
// structure: _data_access_log, used to encapsulate the use history of a shared data,
//           item: it's writers, readers and previous values. It is through this
//           structure that data dependence violation is tracked.
//
struct _data_access_log{
        unsigned int _size;//size of data
        set <int> _readers;
        vector <int> _writers;
        vector <void*> _previous_values; //this will be used to solve anti-dependence
                                         //and output dependence violations (WAR and WAW)
                                         //restoring the data to a value that is safe.

        pthread_mutex_t _data_mutex;


        _data_access_log(){
                pthread_mutex_init (&_data_mutex, NULL);
        };


        template <class T>
        bool _output_violations_are_false(T* data_to_write, int pos){ //this function determines if the output dependence
        //violations are false, that is, if the value to write is the same that is already written by later iterations.
        //Note that in this version this check to avoid unnecessary thread-resetting is only possible for WAW violations, since
        //there is book-keeping of the values that are being written, but not of the values that are being read.

                void* prev_value;
                int first_reader=-2;;
                for (int i=0; i<_writers.size(); i++){
                        if (_writers[i]==-1){
                                prev_value=_previous_values[i];
                        }
                        else if (_writers[i]>pos){
                                if (data_to_write!=(T*)_previous_values[i]){
                                        if (first_reader==-2){
                                                if (_previous_values[i]!=prev_value){
                                                        return false;
                                                }
                                                first_reader=_writers[i];
                                        }
                                        else if (first_reader==_writers[i]){
                                                if (_previous_values[i]!=prev_value){
```

```
                                                return false;
                                        }
                                }
                                else{
                                        return false;
                                }
                        }
                }
        }
        if (first_reader==-2){
                _writers.push_back(pos);
                _previous_values.push_back((void*)data_to_write);
                return true;
        }
        for (int i=0; i<_writers.size(); i++){
                if (_writers[i]==first_reader){
                        _previous_values[i]=(void*)data_to_write;
                }
        }
        _writers.push_back(pos);
        _previous_values.push_back(prev_value);
        return true;

};

vector <int> _cancel_higher_readers(int pos){
//if used before a write_data and the return vector is not empty, a true-dependence violation has ocurred (RAW).
//and the higher readers have to be restarted.
        vector <int> to_cancel;
        set <int>:: iterator it=_readers.begin();
        while ((!_readers.empty()) && it!=_readers.end()){
                if ((*it)>pos){
                        to_cancel.push_back(*it);
                        _readers.erase(*it);
                        it=_readers.begin();
                }
                else {
                        it++;
                }
        }
        if ((!to_cancel.empty())&&(!_writers.empty())){
                for (unsigned int i=0; i<to_cancel.size(); i++){
                        unsigned int j=0;
                        while((!_writers.empty())&& j<_writers.size()){
                                if (to_cancel[i]==_writers[j]){
                                        _writers.erase(_writers.begin()+j); //all the writings of the higher readers on this data
                                        _previous_values.erase(_previous_values.begin()+j);
                                        j=0;
                                }
                                else
                                        j++;
                        }
                }
        }
        return to_cancel;
};

void* _get_previous_value(int pos){
        void* value;
        int currPos=-2;
        //The logic to find the previous value is to get the previous value from the lesser of the higher writers,
        //if that fails, then it is necessary to find the previous value of the latest of the lower writers.

        for (int i=(static_cast<int>(_writers.size())-1); i>-1; i--){ //this order matters because if a thread has written several
                                                //it's first should be restored

                if (static_cast<int>(_writers[i])>pos){
                        if (currPos==-2){
                                value=_previous_values[i];
                                currPos=_writers[i];
                        }
                        else if (static_cast<int>(_writers[i])<=currPos){
                                value=_previous_values[i];
                                currPos=_writers[i];
                        }
                }

        }
        if (currPos==-2){
                for (int i=(static_cast<int>(_writers.size())-1); i>-1; i--){
                        if (static_cast<int>(_writers[i])<=pos){
                                if (currPos==-2){
                                        value=_previous_values[i];
                                        currPos=_writers[i];
                                }
                                else if (static_cast<int>(_writers[i])>currPos){
                                        value=_previous_values[i];
```

```
                                        currPos=_writers[i];
                                }
                        }

                }

        }

        if (currPos!=-2){
                return (void*)value;
        }
        return (void*)-1;

};

vector <int> _cancel_higher_writers(int pos){
//if used on a write function and the return vector is not empty, an output dependence violation has ocurred (WAW) and the
//value has to be restored to it's state before pos, so it can be written over.

//if used on a read function and the return vector is not empty, an anti-dependence violation has ocurred (WAR) and the
//value has to be restored to it's state before pos, so it can be read.
        set <int> to_cancel_set;
        unsigned int i=0;
        while ((!_writers.empty())&& (i<_writers.size())){
                if (_writers[i]>pos){
                        to_cancel_set.insert(_writers[i]);
                        _writers.erase(_writers.begin()+i);
                        _previous_values.erase(_previous_values.begin()+i);
                        i=0;
                }
                else
                        i++;

        }
        vector <int> to_cancel;
        to_cancel.insert(to_cancel.end(), to_cancel_set.begin(), to_cancel_set.end());
        if ((!to_cancel.empty())&&(!_readers.empty())){
                for (unsigned int i=0; i<to_cancel.size(); i++){
                        _readers.erase(to_cancel[i]); //all the readings of the higher writers on this data item have to be unlogg
                }
        }
        return to_cancel;
};
};

//!
//! \class  loop_speculator
//!
//! \brief  implements a class that takes n ordered interations from a loop and
//!         executes them speculatively in parallel, while keeping the sequential
//!         consistency. In order to do this the class relies on write_data and
//!         read_data functions, that check for data dependence violations, as well
//!         as a commit function that helps to verify that an iteration has ended it's
//!         execution.
//!
//!         Additional functionality is provided with functions like append and
//!         valid_til, that allow the pre-loop section to dynamically add new
//!         iterations or substract iterations to an on-going speculative execution.
//!         This allows the model to mimic do-while and do-until behaviour.
//!
//! 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 loop_speculator {
private:

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

        //mutex for synchronized use of the previous group of variables
        pthread_mutex_t _is_active_mutex;

        //thread to manage the pre-loop section, boolean value to check if
        //it has ended it's execution & related mutex
        pthread_t _pl;
```

```
bool _pl_commit;
pthread_mutex_t _pl_mutex;

//the speculative threads and it's data
vector <_loop_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
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-loop section & the speculative threads
void*& _shared_data;

//auxiliary data to reset _shared_data.
int _null_data;

//variable indicating until what iteration the pre-loop section has validated
//the execution (-1 if not used) & related mutex
int _valid_til;
pthread_mutex_t _valid_til_mutex;

//red-black tree that relates the reference of a data element from
//shared_data data with a _data_access_log used to keep track of all the
//possible data dependence violations
map <void*, _data_access_log> _access_log;

//mutex related to the _access_log as a whole
pthread_mutex_t _global_access_log_mutex;

//! private method that allows to cancel speculative threads, whose
//! positions are passed as an argument. It is used if there is a
//! error on starting the speculation, and also for resetting the object.
//!
//! TO BE NOTED:
//! * this method takes all class mutexes. On invocation
//!   _spec_threads_mutex and _global_access_log_mutex should
//!    be on hold and no other class mutex, save those related
//!   to a thread that is not about to be canceled or restarted.
//!
//! * if called with an empty vector, it will reset all logs and
//! wait for the threads in deferred cancel to finish.
//!
//!
void _cancel_spec_threads_and_reset_logs (vector <int> threads_to_delete){

        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]> static_cast<int>(_spec_threads.size())){
                                valid_arguments=false; //one thread has an invalid id.
                                i=threads_to_delete.size();
                        }

                }
        }
        else{
                valid_arguments=false; //no threads to delete

                /*This option will be used to initialize the object, reseting it's inner arrays*/

                pthread_mutex_lock(&_is_active_mutex);

                pthread_mutex_lock(&_valid_til_mutex);

                if (_has_pre_loop)

                        pthread_mutex_lock(&_pl_mutex);

                if (!_spec_threads.empty()){
                        _spec_threads.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_loop)
                                pthread_mutex_unlock(&_pl_mutex);

                        pthread_mutex_unlock(&_valid_til_mutex);

                        pthread_mutex_unlock(&_is_active_mutex);

                        pthread_mutex_unlock(&_spec_threads_mutex);

                        pthread_mutex_unlock(&_global_access_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_access_log_mutex);

                        pthread_mutex_lock(&_spec_threads_mutex);

                        _thread_index.clear();

                        if (!_access_log.empty()){
                                _access_log.clear();
                        }
                }
                else {

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

                        if (_has_pre_loop)

                                pthread_mutex_unlock(&_pl_mutex);

                        pthread_mutex_unlock(&_valid_til_mutex);

                        pthread_mutex_unlock(&_is_active_mutex);

                }

        }


        if (valid_arguments){

                for (unsigned int i=threads_to_delete.size()-1; i>=0; i--){
                        pthread_mutex_lock(&_spec_threads[threads_to_delete[i]]._thread_mutex);
                }

                pthread_t thrd_id=pthread_self();

                pthread_mutex_lock(&_is_active_mutex);
                pthread_mutex_lock(&_valid_til_mutex);

                if (_has_pre_loop)

                        pthread_mutex_lock(&_pl_mutex);

                for (unsigned int i=0; i<threads_to_delete.size(); i++){
                        if (pthread_equal(_spec_threads[threads_to_delete[i]]._thread, thrd_id)==0){
                                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_loop)
                        pthread_mutex_unlock(&_pl_mutex);

                pthread_mutex_unlock(&_valid_til_mutex);

                pthread_mutex_unlock(&_is_active_mutex);

        }
};
```

```
//! private method that allows to cancel speculative threads higher than the
//! one passed as an argument, it is used in the valid_til function.
//!
//! TO BE NOTED:
//! * this method takes all class mutexes. On invocation
//!   _spec_threads_mutex and _global_access_log_mutex should
//!    be on hold and no other class mutex, save those related
//!   to a thread that is not about to be canceled or restarted.
//!
//!
void _cancel_higher_spec_threads (int top_thread_pos){

        bool valid_arguments=false;

        if (top_thread_pos>=-1 && top_thread_pos< static_cast<int>(_spec_threads.size())){

                set <int> threads_to_cancel;
                set <void*> data_to_check;
                int aux_value=top_thread_pos;

                if (aux_value==-1){
                        aux_value=0;
                }
                for (unsigned int i=aux_value; i<_spec_threads.size(); i++){
                        threads_to_cancel.insert(i);
                }

                set <int>::iterator it;
                for (it=threads_to_cancel.begin(); it!=threads_to_cancel.end(); it++){
                        if (*it!=top_thread_pos){
                                pthread_mutex_lock(&_spec_threads[*it]._thread_mutex);
                        }
                }

                set<void*>::iterator it2;
                for (it=threads_to_cancel.begin(); it!=threads_to_cancel.end(); it++){

                        for (it2=_spec_threads[*it]._read_data.begin(); it2!=_spec_threads[*it]._read_data.end(); it2++){

                                if (data_to_check.find(*it2)==data_to_check.end()){

                                        data_to_check.insert(*it2);
                                }
                        }

                        if (*it!=top_thread_pos){

                                _spec_threads[*it]._read_data.clear();

                        }

                        for (it2=_spec_threads[*it]._written_data.begin(); it2!=_spec_threads[*it]._written_data.end(); it2++){

                                if (data_to_check.find(*it2)==data_to_check.end()){

                                        data_to_check.insert(*it2);
                                }
                        }

                        if (*it!=top_thread_pos){

                                _spec_threads[*it]._written_data.clear();

                        }

                }

                pthread_mutex_lock(&_is_active_mutex);
                pthread_mutex_lock(&_valid_til_mutex);
                if (_has_pre_loop)
                        pthread_mutex_lock(&_pl_mutex);

                for (it=threads_to_cancel.begin(); it!=threads_to_cancel.end(); it++){
                        if (*it!=top_thread_pos){
                                _spec_threads[*it]._commit=false;
                                if (pthread_kill(_spec_threads[*it]._thread, 0)==0){
                                        pthread_cancel(_spec_threads[*it]._thread);
                                }
                                _thread_index[_spec_threads[*it]._thread]= -1;
                                pthread_mutex_destroy(&_spec_threads[*it]._thread_mutex);

                        }
                }


                for (it2=data_to_check.begin(); it2!=data_to_check.end(); it2++){
```

```
                        void* new_val= _access_log.find(*it2)->second._get_previous_value(top_thread_pos);
                        vector <int> writers_to_cancel=_access_log.find(*it2)->second._cancel_higher_writers(top_thread_pos);
                        vector <int> readers_to_cancel=_access_log.find(*it2)->second._cancel_higher_readers(top_thread_pos);

                        if (!writers_to_cancel.empty()){
                                memcpy((void*&)const_cast<void*&>(*it2), (void*)&new_val, _access_log.find(*it2)->second._size);
                        }

                }
                if (top_thread_pos==-1){
                        _spec_threads.clear();
                }
                else {
                        _spec_threads.resize(top_thread_pos+1);
                }

                if (_has_pre_loop)
                        pthread_mutex_unlock(&_pl_mutex);
                pthread_mutex_unlock(&_valid_til_mutex);
                pthread_mutex_unlock(&_is_active_mutex);

        }
};

//! private method that allows to reset speculative threads, which has to be called
//! from a thread whose position is passed as argument. This method is used for reseting
//! threads that commited some form of data dependence violation.
//!
//! TO BE NOTED:
//! * this method takes all class mutexes. On invocation
//!   _spec_threads_mutex and _global_access_log_mutex should
//!    be on hold and no other class mutex, save those related
//!   to a thread that is not about to be canceled or restarted.
//!
//!
void _reset_spec_threads (int current_thread, void* data_currently_held, vector <int> threads_to_reset){

        int i;
        bool valid_arguments=true;

        if (!threads_to_reset.empty()){

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

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

                }
        }
        else{
                valid_arguments=false; //no threads to delete
        }
        if (current_thread<-1){
                valid_arguments=false;
        }

        if (valid_arguments){

                set <int> threads_currently_held;
                threads_currently_held.insert(current_thread);

                set <int> set_of_threads_to_reset;
                set_of_threads_to_reset.insert(threads_to_reset.begin(), threads_to_reset.end());

                set<int>:: iterator it;
                it=set_of_threads_to_reset.begin();

                set <void*> data_already_checked;
                data_already_checked.insert(data_currently_held);

                while (it!=set_of_threads_to_reset.end()){

                        unsigned int previous_size=set_of_threads_to_reset.size();

                        if (threads_currently_held.find(*it)==threads_currently_held.end()){

                                pthread_mutex_lock(&_spec_threads[*it]._thread_mutex);
                                threads_currently_held.insert(*it);

                                set<void*>::iterator it2;

                                for (it2=_spec_threads[*it]._read_data.begin(); it2!=_spec_threads[*it]._read_data.end(); it2++){

                                        _access_log.find(*it2)->second._readers.erase(*it);
                                }
```

```
                        _spec_threads[*it]._read_data.clear();


                        for (it2=_spec_threads[*it]._written_data.begin(); it2!=_spec_threads[*it]._written_data.end(); it
                                if (data_already_checked.find(const_cast<void*&>(*it2))==data_already_checked.end()){

                                        data_already_checked.insert(const_cast<void*&>(*it2));

                                        void* value_to_restore= _access_log.find(*it2)->second._get_previous_value(current

                                        vector <int> writers_to_cancel=_access_log.find(*it2)->second._cancel_higher_write

                                        vector <int> readers_to_cancel=_access_log.find(*it2)->second._cancel_higher_reade

                                        if (!writers_to_cancel.empty()){//implied or secondary WAW and WAR violations have
                                                memcpy((void*&)const_cast<void*&>(*it2), (void*)&value_to_restore, _access
                                        }

                                        if (!readers_to_cancel.empty()){//implied or secondary RAW violations have ocurred
                                                writers_to_cancel.insert(writers_to_cancel.end(),readers_to_cancel.begin()
                                        }
                                        if (!writers_to_cancel.empty()){
                                                for (unsigned int i=0; i<writers_to_cancel.size(); i++){
                                                        set_of_threads_to_reset.insert(writers_to_cancel[i]);
                                                }
                                        }

                                }
                        }
                        _spec_threads[*it]._written_data.clear();

                        if ((*it!=current_thread)&&(pthread_equal(_spec_threads[*it]._thread, pthread_self())==0)){

                                pthread_mutex_lock(&_is_active_mutex);
                                pthread_mutex_lock(&_valid_til_mutex);
                                if (_has_pre_loop)
                                        pthread_mutex_lock(&_pl_mutex);

                                _spec_threads[*it]._commit=false;
                                if (pthread_kill(_spec_threads[*it]._thread, 0)==0){
                                        pthread_cancel(_spec_threads[*it]._thread);
                                }
                                _thread_index[_spec_threads[*it]._thread]= -1;

                                if (_has_pre_loop)
                                        pthread_mutex_unlock(&_pl_mutex);
                                pthread_mutex_unlock(&_valid_til_mutex);
                                pthread_mutex_unlock(&_is_active_mutex);

                        }



                }
                if (set_of_threads_to_reset.size()>previous_size){
                        it=set_of_threads_to_reset.begin();
                }
                else {
                        it++;
                }
        }

        pthread_mutex_lock(&_is_active_mutex);
        pthread_mutex_lock(&_valid_til_mutex);
        if (_has_pre_loop)
                pthread_mutex_lock(&_pl_mutex);

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


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

                if (*it!=current_thread) {

                        if (_valid_til!=-1){
                                if (*it<=_valid_til){
                                        pthread_create(&_spec_threads[*it]._thread, &attr, _spec_threads[*it]._thread_inst
                                        _thread_index[_spec_threads[*it]._thread]= *it;
                                }

                        }
                        else {
                                pthread_create(&_spec_threads[*it]._thread, &attr, _spec_threads[*it]._thread_instructions
                                _thread_index[_spec_threads[*it]._thread]= *it;
```

```
                                        }
                                }
                        }

                        if (_has_pre_loop)
                                pthread_mutex_unlock(&_pl_mutex);

                        pthread_mutex_unlock(&_valid_til_mutex);
                        pthread_mutex_unlock(&_is_active_mutex);

                        for (it=threads_currently_held.begin(); it!=threads_currently_held.end(); it++){
                                if (*it!=current_thread)
                                        pthread_mutex_unlock(&_spec_threads[*it]._thread_mutex);
                        }
                }
        };


public:

        //!
        //! default constructor
        //!
        loop_speculator():_shared_data((void*&)_null_data){

                _is_active=false;
                _pl_commit=false;
                _has_pre_loop=false;
                _valid_til=-1;//the valid til funcionality is not used
                _null_data=-1;
                pthread_mutex_init (&_is_active_mutex, NULL);
                pthread_mutex_init (&_pl_mutex, NULL);
                pthread_mutex_init (&_spec_threads_mutex, NULL);
                pthread_mutex_init (&_valid_til_mutex, NULL);
                pthread_mutex_init (&_global_access_log_mutex, NULL);

        };


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

                pthread_mutex_lock(&_is_active_mutex);

                bool manages_pre_loop=_has_pre_loop;
                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 thread or thread in deferred cancelation
                }
                pthread_mutex_unlock(&_spec_threads_mutex);


                if (manages_pre_loop){

                        pthread_mutex_lock(&_pl_mutex);

                        if (pthread_self()==_pl){

                                pthread_mutex_unlock(&_pl_mutex);

                                return _shared_data;
                        }

                        else {
                                pthread_mutex_unlock(&_pl_mutex);

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

```
        return _shared_data; //unmanaged pre-loop or thread in deferred cancelation
};


//!
//! required function that signals that a given thread has ended
//! it's execution
//!
void commit (){

        pthread_mutex_lock(&_is_active_mutex);

        if (_is_active){

                bool manages_pre_loop=_has_pre_loop;
                pthread_mutex_unlock(&_is_active_mutex);

                pthread_mutex_lock(&_spec_threads_mutex);


                if (!_thread_index.empty()){ //thread in deferred cancel.
                        pthread_t thread_id=pthread_self();

                        if (_thread_index.find(thread_id)!=_thread_index.end()){

                                int pos=_thread_index.find(thread_id)->second;
                                if (pos>=0){ //valid thread
                                        pthread_mutex_lock(&_spec_threads[pos]._thread_mutex);
                                        _spec_threads[pos]._commit=true;
                                        pthread_mutex_unlock(&_spec_threads[pos]._thread_mutex);
                                }
                        }
                        else if (manages_pre_loop){
                                if (thread_id==_pl){
                                        pthread_mutex_lock(&_pl_mutex);
                                        _pl_commit=true;
                                        pthread_mutex_unlock(&_pl_mutex);
                                }
                        }
                        else {  //unmanaged pre-loop
                                pthread_mutex_lock(&_pl_mutex);
                                _pl_commit=true;
                                pthread_mutex_unlock(&_pl_mutex);
                        }
                }

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

};

//!
//! function to be called from the speculative threads
//! and the concurrent pre-loop section, 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){

                bool manages_pre_loop=_has_pre_loop;
                pthread_mutex_unlock(&_is_active_mutex);

                int current_thread=-2;

                pthread_mutex_lock(&_spec_threads_mutex);

                pthread_t thread_id=pthread_self();

                if (_thread_index.empty()){//thread in deferred cancel.
                        current_thread=-3;
                }
                else if (_thread_index.find(thread_id)!=_thread_index.end()){
                        current_thread=_thread_index.find(thread_id)->second;
                        if (current_thread==-1)//thread in deferred cancel.
                                current_thread=-3;
                }
                else {
                        if (manages_pre_loop){
                                pthread_mutex_lock(&_pl_mutex);
```

```
                if (thread_id==_pl){
                        current_thread=-1;
                }
                else {
                        current_thread=-3; //invalid caller
                }
                pthread_mutex_unlock(&_pl_mutex);
        }
        else {
                current_thread=-1;
        }
}
if (current_thread>=-1){//is a valid read.

        T retval;

        pthread_mutex_lock(&_valid_til_mutex);

        if (_valid_til!=-1 && _valid_til<current_thread){
                retval=*(T*)data_to_be_read;
                pthread_mutex_unlock(&_valid_til_mutex);
                pthread_mutex_unlock(&_spec_threads_mutex);
                return retval; //thread that is no longer valid.
        }
        pthread_mutex_unlock(&_valid_til_mutex);

        pthread_mutex_lock(&_global_access_log_mutex);
        if (_access_log.find((void*)data_to_be_read)==_access_log.end()){
        //if the data has never been accesed, it's log should be created

                pthread_mutex_t temp;
                pthread_mutex_init (&temp, NULL);
                _data_access_log temp2;

                //since it's the data's first use, it's assumed to be the value in the pre-loop
                //so the data is initialized on that premise
                temp2._size=sizeof(T);
                temp2._writers.push_back((int)-1);
                temp2._previous_values.push_back((void*)*(T*)data_to_be_read);

                if (current_thread!=-1){
                        pthread_mutex_lock(&_spec_threads[current_thread]._thread_mutex);
                        _spec_threads[current_thread]._read_data.insert((void*)data_to_be_read);
                        temp2._readers.insert(current_thread);//the current read is logged
                }

                pthread_mutex_unlock(&_spec_threads_mutex);
                _access_log.insert (pair<void*, _data_access_log>((void*)data_to_be_read, temp2));
                retval=*(T*)data_to_be_read;

                if (current_thread!=-1){
                        pthread_mutex_unlock(&_spec_threads[current_thread]._thread_mutex);
                }

                pthread_mutex_unlock(&_global_access_log_mutex);
                return retval;
        }

        _data_access_log* ptr=&_access_log.find((void*)data_to_be_read)->second;
        pthread_mutex_lock(&ptr->_data_mutex);
        pthread_mutex_unlock(&_global_access_log_mutex);

        retval= (T) ptr->_get_previous_value(current_thread);
        vector <int> threads_to_cancel= ptr->_cancel_higher_writers(current_thread);


        //if the log exists, then anti-dependence violation is checked (WAR)

        if (current_thread!=-1){
                pthread_mutex_lock(&_spec_threads[current_thread]._thread_mutex);
        }


        if (!threads_to_cancel.empty()){//an anti-dependence violation has ocurred
                memcpy((void*)data_to_be_read, (void*)&retval, sizeof(T)); //the previous value is restored.
                _reset_spec_threads(current_thread, (void*)data_to_be_read, threads_to_cancel);
        }
        else{
                retval=*(T*) data_to_be_read;//no anti-dependence violation has ocurred, the data can be read on i
        }
        pthread_mutex_unlock(&_spec_threads_mutex);

        //now that the reading is done, it has to be logged
        if (current_thread!=-1){
                ptr->_readers.insert(current_thread);
                _spec_threads[current_thread]._read_data.insert((void*)data_to_be_read);
```

```
                                pthread_mutex_unlock(&_spec_threads[current_thread]._thread_mutex);
                        }
                        pthread_mutex_unlock(&ptr->_data_mutex);
                        return retval;
                }
                pthread_mutex_unlock(&_spec_threads_mutex);
        }
        else {
                pthread_mutex_unlock(&_is_active_mutex);
        }
        return *(T*) data_to_be_read; //invalid caller, inactive object or thread in deferred cancel.
                                      //it's calling data is returned instead of NULL, to prevent segmentation faults.
};

//!
//! function to be called from the speculative threads
//! and concurrent pre-loop section, 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){
                bool manages_pre_loop=_has_pre_loop;
                pthread_mutex_unlock(&_is_active_mutex);

                int current_thread=-2;

                pthread_mutex_lock(&_spec_threads_mutex);

                pthread_t thread_id=pthread_self();

                if (_thread_index.empty()){//thread in deferred cancel.
                        current_thread=-3;
                }
                else if (_thread_index.find(thread_id)!=_thread_index.end()){
                        current_thread=_thread_index.find(thread_id)->second;
                        if (current_thread==-1)//thread in deferred cancel.
                                current_thread=-3;
                }
                else {
                        if (manages_pre_loop){
                                pthread_mutex_lock(&_pl_mutex);
                                if (thread_id==_pl){
                                        current_thread=-1;
                                }
                                else {
                                        current_thread=-3; //invalid caller
                                }
                                pthread_mutex_unlock(&_pl_mutex);
                        }
                        else {
                                current_thread=-1;
                        }
                }
                if (current_thread>=-1){
                        pthread_mutex_lock(&_valid_til_mutex);
                        if (_valid_til!=-1 && _valid_til<current_thread){
                                pthread_mutex_unlock(&_valid_til_mutex);
                                pthread_mutex_unlock(&_spec_threads_mutex);
                                return -1;
                        }
                        pthread_mutex_unlock(&_valid_til_mutex);


                        pthread_mutex_lock(&_global_access_log_mutex);
                        if (_access_log.find((void*)data_to_be_written_upon)==_access_log.end()){
                        //if data has no log entry, then it's entry should be created.


                                pthread_mutex_t temp;
                                pthread_mutex_init (&temp, NULL);
                                _data_access_log temp2;

                                //since it's the data's first use, it's assumed to be the value in the pre-loop
                                //so the data is initialized on that premise
                                temp2._size=sizeof(T);
                                temp2._writers.push_back((int)-1);
                                temp2._previous_values.push_back((void*)*(T*)data_to_be_written_upon);

                                if (current_thread!=-1){
                                        pthread_mutex_lock(&_spec_threads[current_thread]._thread_mutex);
                                        _spec_threads[current_thread]._written_data.insert((void*)data_to_be_written_upon);
                                        temp2._writers.push_back(current_thread);
                                        temp2._previous_values.push_back((void*)*(T*)data_to_be_written_upon);
                                }
```

```
                else{
                        temp2._writers.push_back(-1);
                        temp2._previous_values.push_back((void*)data_to_write);
                }
                pthread_mutex_unlock(&_spec_threads_mutex);

                _access_log.insert (pair<void*, _data_access_log>((void*)data_to_be_written_upon, temp2));

                //and finally the writting is done
                memcpy ((void*)data_to_be_written_upon, (void*)&data_to_write, sizeof(T));

                if (current_thread!=-1){
                        pthread_mutex_unlock(&_spec_threads[current_thread]._thread_mutex);
                }
                pthread_mutex_unlock(&_global_access_log_mutex);
                return 0;
        }

//if the log exists, then possible true dependence and output dependence violations are checked (RAW and WAW)

        _data_access_log* ptr=&_access_log.find((void*)data_to_be_written_upon)->second;
        pthread_mutex_lock(&ptr->_data_mutex);
        pthread_mutex_unlock(&_global_access_log_mutex);

        if (current_thread!=-1){
                pthread_mutex_lock(&_spec_threads[current_thread]._thread_mutex);
        }

        T value;
        vector <int> writers_to_cancel;

        bool no_output_violation=false;
        if ((void*)*(T*)data_to_be_written_upon==(void*)data_to_write){
        //Since restarting threads has such a significant cost, it is better to check if it is really necessary to
        //So, we will evaluate false positives in output dependence violations.

                if (!ptr->_output_violations_are_false(data_to_write, current_thread)){

                value= (T) ptr->_get_previous_value(current_thread);
                writers_to_cancel=ptr->_cancel_higher_writers(current_thread);

                }
                else {
                        no_output_violation=true;
                }
        }
        else{
                value= (T) ptr->_get_previous_value(current_thread);
                writers_to_cancel=ptr->_cancel_higher_writers(current_thread);

        }
        vector <int> readers_to_cancel=ptr->_cancel_higher_readers(current_thread);

        if (no_output_violation && readers_to_cancel.empty()){
                if (current_thread!=-1){
                        _spec_threads[current_thread]._written_data.insert((void*)data_to_be_written_upon);
                        pthread_mutex_unlock(&_spec_threads[current_thread]._thread_mutex);
                }
                pthread_mutex_unlock(&ptr->_data_mutex);
                pthread_mutex_unlock(&_spec_threads_mutex);
                return 0;
        }

        if (!writers_to_cancel.empty()){
        //an output dependence violation has ocurred, it's previous value has to be restored before writting over,
        //those writers have to be restarted
                ptr->_writers.push_back(current_thread);
                ptr->_previous_values.push_back((void*)value);
        }

        /*This branch has been turned into a comment, since the logging for all cases where there are false output
        is done on _output_violations_are_false(..). However, should a unforseeable case arise where this does not
        (for instance, a particular thread reset pattern), then it should be un-commented.*/

        else if (!no_output_violation){
                //no output dependence violation, it's current value is valid and logged as the previous value of
                //thread write...
                ptr->_writers.push_back(current_thread);
                ptr->_previous_values.push_back((void*)*(T*)data_to_be_written_upon);
        }

        if (!readers_to_cancel.empty()){//a true dependence violation has ocurred, those readers should be restart
                writers_to_cancel.insert(writers_to_cancel.end(), readers_to_cancel.begin(), readers_to_cancel.end
        }
        if (!writers_to_cancel.empty()){
                //the violating threads are restarted
```

```
                                _reset_spec_threads(current_thread, (void*) data_to_be_written_upon, writers_to_cancel);
                        }
                        pthread_mutex_unlock(&_spec_threads_mutex);

                        //now some more logging of the write
                        if (current_thread==-1){
                                ptr->_writers.push_back(-1);
                                ptr->_previous_values.push_back((void*)data_to_write);
                        }
                        else {
                                _spec_threads[current_thread]._written_data.insert((void*)data_to_be_written_upon);
                        }

                        //and finally the write is made
                        memcpy ((void*)data_to_be_written_upon, (void*)&data_to_write, sizeof(T));

                        if (current_thread!=-1){
                                pthread_mutex_unlock(&_spec_threads[current_thread]._thread_mutex);
                        }
                        pthread_mutex_unlock(&ptr->_data_mutex);
                        return 0;
                }
                pthread_mutex_unlock(&_spec_threads_mutex);
        }
        else {
                pthread_mutex_unlock(&_is_active_mutex);
        }
        return -1; //invalid caller, thread in deferred cancel or inactive object.
};

//!
//! function that permits the pre_loop section 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_loop=_has_pre_loop;
        pthread_mutex_unlock(&_is_active_mutex);

        bool is_pre_loop=false;

        pthread_mutex_lock(&_spec_threads_mutex);

        if (manages_pre_loop){
                pthread_mutex_lock(&_pl_mutex);
                if (pthread_self()==_pl){
                        is_pre_loop=true;
                }
                pthread_mutex_unlock(&_pl_mutex);
        }
        if (!is_pre_loop){
                pthread_t thrd_id=pthread_self();
                if ((!_thread_index.empty()) && (_thread_index.find(thrd_id)==_thread_index.end())){
                        if (!manages_pre_loop){
                                is_pre_loop=true;
                        }
                }

        }
        if (is_pre_loop){

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

                int current_size=_spec_threads.size();

                pthread_mutex_lock(&_valid_til_mutex);
                if (_valid_til!=-1 && (current_size-1)>=_valid_til){
                        pthread_mutex_unlock(&_valid_til_mutex);
                        pthread_mutex_unlock(&_spec_threads_mutex);
                        return -1;//the thread to append does not fit with the valid_til condition
                }
                pthread_mutex_unlock(&_valid_til_mutex);

                _spec_threads.resize(_spec_threads.size()+1);
                pthread_mutex_lock(&_spec_threads[_spec_threads.size()-1]._thread_mutex);
                int i=_spec_threads.size()-1;
                _spec_threads[i]._commit=false;
                _spec_threads[i]._thread_instructions=f;
                _spec_threads[i]._const_args=consts;
```

```
                success=pthread_create(&_spec_threads[i]._thread, &attr, f, consts);
                if (success!=0){
                        pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);
                        _spec_threads.resize(_spec_threads.size()-1);
                        pthread_mutex_unlock(&_spec_threads_mutex);
                        return -1;//the thread could not be created, thus is not appended
                }
                _thread_index.insert(pair<pthread_t, int>(_spec_threads[_spec_threads.size()-1]._thread, _spec_threads.size()-1));
                pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);
                pthread_mutex_unlock(&_spec_threads_mutex);
                return 0;
        }
        pthread_mutex_unlock(&_spec_threads_mutex);
        return -1; //invalid caller
};


//!
//! function that allows the pre_loop section to dynamically cancel
//! a number of speculative threads at the end of the array in an on-going
//! speculative execution, this is done by indicating until what position
//! the execution remains valid.
//!
//! TO BE NOTED: It can only be called once during all the execution.
//!
int valid_til (int pos){
        if (pos<-1)
                return -1;//invalid argument

        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_loop=_has_pre_loop;
        pthread_mutex_unlock(&_is_active_mutex);

        pthread_mutex_lock(&_valid_til_mutex);
        if (_valid_til!=-1){
                pthread_mutex_unlock(&_valid_til_mutex);
                return -1; //the valid_til function has already been called
        }
        pthread_mutex_unlock(&_valid_til_mutex);

        bool is_pre_loop=false;

        pthread_mutex_lock(&_spec_threads_mutex);

        if (manages_pre_loop){
                pthread_mutex_lock(&_pl_mutex);
                if (pthread_self()==_pl){
                        is_pre_loop=true;
                }
                pthread_mutex_unlock(&_pl_mutex);
        }
        if (!is_pre_loop){
                pthread_t thrd_id=pthread_self();
                if ((!_thread_index.empty()) && (_thread_index.find(thrd_id)==_thread_index.end())){
                        if (!manages_pre_loop){
                                is_pre_loop=true;
                        }
                }

        }

        if (is_pre_loop){
                pthread_mutex_lock(&_valid_til_mutex);
                _valid_til=pos;
                pthread_mutex_unlock(&_valid_til_mutex);

                if ( (static_cast<int>(_spec_threads.size())-1)>pos){
                        _cancel_higher_spec_threads (pos);

                }
                pthread_mutex_unlock(&_spec_threads_mutex);
                return 0; //no thread was canceled, the restriction will be
                          //imposed through the append function
        }
        pthread_mutex_unlock(&_spec_threads_mutex);
        return -1;//invalid caller
};

//!
//! speculate: a complete function that takes orderly the instructions and arguments
//! of the pre-loop section, as well as those of the loop iterations; 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-
//!    loop section. This means that on invocation, the caller blocks until the pre-
//!    loop and all the iterations up to the end or valid_til have commited.
//!    * If the pre-loop section does not explicitly commit through the commit function,
//!    then the whole results of the speculation will not be communicated to the
//!    shared data and the speculation will be canceled.
//!    * If any iteration does not commit, then only the results up to it will be taken
//!    into account, the remaining iterations will be canceled.
//!
int speculate (void*& shared_data, void* (fpl)(void*), void* const_args_pl, script_vector thread_instructions, vector <void*> cons
        if (thread_instructions.size()!=const_args.size()){
                return -1; //invalid parameters
        }
        pthread_mutex_lock(&_is_active_mutex);
        if (_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1; //the object is active
        }
        _is_active=true;
        pthread_mutex_unlock(&_is_active_mutex);


        pthread_mutex_lock(&_spec_threads_mutex);
        pthread_mutex_lock(&_global_access_log_mutex);

        vector <int> threads_to_cancel;//a selective reset is performed, waiting for the threads in
        _cancel_spec_threads_and_reset_logs(threads_to_cancel);//deferred cancel to finish, and reseting the
        //arrays that the speculation uses.

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

        pthread_mutex_lock(&_valid_til_mutex);
        _valid_til=-1;
        pthread_mutex_unlock(&_valid_til_mutex);

        pthread_mutex_lock(&_pl_mutex);
        _pl_commit=false;
        pthread_mutex_unlock(&_pl_mutex);

        _shared_data=shared_data;

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

        //the threads are created
        int success;
        int valid_til_thread=-1;
        _spec_threads.resize(thread_instructions.size());
        pthread_mutex_lock(&_pl_mutex);
        success=pthread_create (&_pl, &attr, fpl, const_args_pl);
        pthread_mutex_unlock(&_pl_mutex);
        if (success!=0){
                _spec_threads.clear();
                pthread_mutex_unlock(&_global_access_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-loop section could not be created
        }

        for (unsigned int i=0; i<thread_instructions.size(); 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[i];
                success=pthread_create (&_spec_threads[i]._thread, &attr, _spec_threads[i]._thread_instructions, _spec_threads[i].

                if (success==0){
                        _thread_index.insert(pair<pthread_t, int>(_spec_threads[i]._thread, i));
                }

                pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);

                if (success!=0){
                        for (unsigned int j=i+1; j<thread_instructions.size(); j++){
```

```
                        pthread_mutex_unlock(&_spec_threads[j]._thread_mutex);
                }
                for (unsigned int j=0; j<i; j++){
                        threads_to_cancel.push_back(j);
                }
                _cancel_spec_threads_and_reset_logs(threads_to_cancel);


                pthread_mutex_unlock(&_global_access_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_access_log_mutex);
pthread_mutex_unlock(&_spec_threads_mutex);


bool commit_made=false;
int joined_but_didnt_commit=0;
do{

        success=pthread_join (_pl, NULL);

        if (success!=0){
                pthread_mutex_lock(&_pl_mutex);
                if (_pl_commit){
                        commit_made=true;
                }
                else{
                        if (joined_but_didnt_commit<10){
                                success=0;
                                joined_but_didnt_commit++;
                        }
                }


                /*just in case the commit is not logged in a timely fasion, it is tried
                10 times to see if it get's made, if not then the object asumes there was no commitment in the pre-loop.
                Although slightly un-elegant, the validity of this solution has not been disproved by empirical
                tests.*/
                pthread_mutex_unlock(&_pl_mutex);
        }


} while (success==0);

pthread_mutex_lock(&_spec_threads_mutex);


if (!commit_made){
                _cancel_higher_spec_threads(-1);
                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-loop did not commit.
}


int i=0;

while (_spec_threads.size()>i){
        pthread_mutex_unlock(&_spec_threads_mutex);

        commit_made=false;
        joined_but_didnt_commit=0;

        do{

                success=pthread_join (_spec_threads[i]._thread, NULL);

                if (success!=0){
                        pthread_mutex_lock(&_spec_threads[i]._thread_mutex);
                        if (_spec_threads[i]._commit){
                                commit_made=true;
                        }
                        else{
                                if (joined_but_didnt_commit<100){
                                        success=0;
                                        joined_but_didnt_commit++;
                                }
```

```
                                }
                                pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);
                        }


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

                } while (success==0);

                if (!commit_made){
                        pthread_mutex_lock(&_spec_threads_mutex);
                        _cancel_higher_spec_threads(i); //This is perhaps a bit excesive, but for consistency purposes it is manda
                        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-loop did not commit.
                }

                i++;

                pthread_mutex_lock(&_spec_threads_mutex);

                pthread_mutex_lock(&_valid_til_mutex);
                valid_til_thread=_valid_til;
                pthread_mutex_unlock(&_valid_til_mutex);

                if (valid_til_thread!=-1 && i>valid_til_thread){
                        i=_spec_threads.size();
                }

        }
        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 complete function that takes orderly the instructions and arguments
//! of the loop iterations; and starts their speculatively parallel execution, while
//! maintaining the sequential consistency with an un-managed pre-loop section.
//! This function will return in the &shared_data, the results of the computation, when
//! get_results is called.
//!
//! TO BE NOTED:
//!    * In this version of the function, the object is not in control of the pre-
//!    loop section. This means that on invocation, the caller only blocks for the creation
//!    of the threads, and can resume it's execution as a possible pre-loop, until
//!    calling get_results(), when the caller blocks until all valid loop iterations/threads return.
//!
int speculate (void*& shared_data, script_vector thread_instructions, vector <void*> const_args){
        if (thread_instructions.size()!=const_args.size()){
                return -1; //invalid parameters
        }
        pthread_mutex_lock(&_is_active_mutex);
        if (_is_active){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1; //the object is active
        }
        _is_active=true;
        pthread_mutex_unlock(&_is_active_mutex);

        pthread_mutex_lock(&_spec_threads_mutex);
        pthread_mutex_lock(&_global_access_log_mutex);

        vector <int> threads_to_cancel;//a selective reset is performed, waiting for the threads in
        _cancel_spec_threads_and_reset_logs(threads_to_cancel);//deferred cancel to finish, and reseting the
        //arrays that the speculation uses.

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

        pthread_mutex_lock(&_valid_til_mutex);
        _valid_til=-1;
        pthread_mutex_unlock(&_valid_til_mutex);
```

```
        pthread_mutex_lock(&_pl_mutex);
        _pl_commit=false;
        pthread_mutex_unlock(&_pl_mutex);

        _shared_data=shared_data;

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

        //the threads are created
        int success;
        _spec_threads.resize(thread_instructions.size());

        for (unsigned int i=0; i<thread_instructions.size(); 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[i];
                success=pthread_create (&_spec_threads[i]._thread, &attr, _spec_threads[i]._thread_instructions, _spec_threads[i].

                if (success==0){
                        _thread_index.insert(pair<pthread_t, int>(_spec_threads[i]._thread, i));
                }

                pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);

                if (success!=0){
                        for (unsigned int j=i+1; j<thread_instructions.size(); j++){
                                pthread_mutex_unlock(&_spec_threads[j]._thread_mutex);
                        }
                        for (unsigned int j=0; j<i; j++){
                                threads_to_cancel.push_back(j);
                        }
                        _cancel_spec_threads_and_reset_logs(threads_to_cancel);

                        pthread_mutex_unlock(&_global_access_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_access_log_mutex);
        pthread_mutex_unlock(&_spec_threads_mutex);


        return 0;
};

//!
//! function to get the results of the speculation once the un-managed pre-
//! loop section has ended it's execution.
//!
//! TO BE NOTED:
//!    * If the pre-loop section does not explicitly commit through the commit function,
//!    then the whole results of the speculation will not be communicated to the
//!    shared data and the speculation will be canceled.
//!    * If any iteration does not commit, then the object will be in an infinite loop wa-
//!   iting for it to commit.
int get_results(){
        pthread_mutex_lock(&_is_active_mutex);
        if (!_is_active || _has_pre_loop){
                pthread_mutex_unlock(&_is_active_mutex);
                return -1;//the object is inactive or manages it's pre-loop section, thus this function cannot be run.
        }
        pthread_mutex_unlock(&_is_active_mutex);

        bool commit_made=false;
        unsigned int joined_but_didnt_commit=0;
        int success;

        do{

                success=-1;

                if (success!=0){
                        pthread_mutex_lock(&_pl_mutex);
                        if (_pl_commit){
                                commit_made=true;
                        }
                        else{
```

```
                        if (joined_but_didnt_commit<10){
                                success=0;
                                joined_but_didnt_commit++;
                        }
                }
                pthread_mutex_unlock(&_pl_mutex);
        }

} while (success==0);

int i=0;


pthread_mutex_lock(&_spec_threads_mutex);


if (!commit_made){
                _cancel_higher_spec_threads(-1);
                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-loop did not commit.
}

while (_spec_threads.size()>i){

        pthread_t thread_to_join=_spec_threads[i]._thread;

        pthread_mutex_unlock(&_spec_threads_mutex);

        commit_made=false;
        joined_but_didnt_commit=0;


        do{

                success=pthread_join (thread_to_join, NULL);

                if (success!=0){
                        pthread_mutex_lock(&_spec_threads[i]._thread_mutex);
                        if (_spec_threads[i]._commit){
                                commit_made=true;
                        }
                        else{
                                if (joined_but_didnt_commit<100){
                                        success=0;
                                        joined_but_didnt_commit++;
                                }
                        }
                        pthread_mutex_unlock(&_spec_threads[i]._thread_mutex);
                }


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

        } while (success==0);

        if (!commit_made){

                pthread_mutex_lock(&_spec_threads_mutex);
                _cancel_higher_spec_threads(i); //This is perhaps a bit excesive, but for consistency purposes it is manda
                pthread_mutex_unlock(&_spec_threads_mutex);

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

                return -1; //some thread did not commit.
        }

        i++;

        pthread_mutex_lock(&_spec_threads_mutex);

        pthread_mutex_lock(&_valid_til_mutex);
        int valid_til_thread=_valid_til;
        pthread_mutex_unlock(&_valid_til_mutex);

        if (valid_til_thread!=-1 && i>valid_til_thread){
                i=_spec_threads.size();
        }
```

```
                }

                pthread_mutex_unlock(&_spec_threads_mutex);

                pthread_mutex_lock(&_is_active_mutex);
                _is_active=false;
                pthread_mutex_unlock(&_is_active_mutex);
                return 0;
        };


};
```