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

//!
//! \class  critical_section_speculator
//!
//! \brief  implements a class that allows multiple threads to process concurrently shared,
//!         data, while keeping a sequential consistency related to the order in which this
//!         threads invoque the speculation. This order is user's responsibility.
//!
//!         Through this class instead of using locks, each thread can go speculatively into
//!         it's critical section, asuming that it will consume the values produced by the
//!         previous threads affecting the data and none of the values produced by later threads.
//!
//!         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 per-thread required
//!         commit function, with which the thread signals that it has ended it's use of
//!         the critical section.
//!
//! 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 critical_section_speculator {
private:

        //bool values to set if the speculator is active or if the shared_data
        //has already been initialized.
        bool _is_active, _has_fixed_shared_data;

        //mutex for synchronized use of the previous variable
        pthread_mutex_t _is_active_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 threads
        void*& _shared_data;

        //auxiliary data to reset _shared_data.
        int _null_data;

        //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 reset 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.
        //!
        //!
        void _reset_speculator (){
```

```
		if (!_spec_threads.empty()){
			for (int i=0; i<_spec_threads.size(); i++){
				pthread_mutex_lock(&_spec_threads[i]._thread_mutex);
			}
		}

		pthread_mutex_lock(&_is_active_mutex);

		if (!_spec_threads.empty()){
			for (int i=0; i<_spec_threads.size(); i++){
				pthread_mutex_destroy(&_spec_threads[i]._thread_mutex);
			}
			_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, affecting its future behaviour*/

			_is_active=false;

			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();
			}
			_is_active=false;

			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, 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<0){
                        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();

                        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

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

                                                vector <int> writers_to_cancel=_access_log.find(*it2)->second._cancel_higher_writers(curre
                                                vector <int> readers_to_cancel=_access_log.find(*it2)->second._cancel_higher_readers(curre

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

                                                }
                                                if (!readers_to_cancel.empty()){//implied or secondary RAW violations have ocurred
                                                        writers_to_cancel.insert(writers_to_cancel.end(),readers_to_cancel.begin(),readers
                                                }
                                                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);

                                                _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_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_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) {

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

			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
	//!
	critical_section_speculator():_shared_data((void*&)_null_data){

		_is_active=false;
		_null_data=-1;
		_has_fixed_shared_data=false;
		pthread_mutex_init (&_is_active_mutex, NULL);
		pthread_mutex_init (&_spec_threads_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);

		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);


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

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

		pthread_mutex_lock(&_is_active_mutex);

		if (_is_active){

			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);
                                }
                        }
                }

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

};


//!
//! function to be called from the speculative threads
//! 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);

                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 {
                        current_thread=-3; //invalid caller
                }
                if (current_thread>=-1){//is a valid read.

                        T retval;

                        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-critical-section
                                //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, 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
        //! 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);

                        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 {
                                current_thread=-3; //invalid caller
                        }
                        if (current_thread>=-1){

                                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 initial value
                                        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 tc
                                //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, 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.
};


//!
//! speculate: a function that takes the instructions and arguments of a critical section
//! of a given thread and starts it's execution alongside with other threads in their
//! own related critical sections, while maintaining the sequential consistency.
//! This function modifies the global &shared_data, and returns a void*, representing
//! additional per-thread values the user might return from the function given as argument.
//!
//! TO BE NOTED: This function waits in an infinite loop for the threads to signal that they
//! have commited.
//!
void* speculate (void*& shared_data, void* (f)(void*), void* const_args_f){


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

        pthread_mutex_lock(&_is_active_mutex);
        if (!_is_active){
                _is_active=true;
                _has_fixed_shared_data=false;
        }
        pthread_mutex_unlock(&_is_active_mutex);

        if (!_has_fixed_shared_data){
                _shared_data=shared_data;
                _has_fixed_shared_data=true;//if the speculator was resetted, the shared data is re-loaded.
        }

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

        //the thread is created
        int pos=_spec_threads.size();
        _spec_threads.resize(pos+1);
        pthread_mutex_lock(&_spec_threads[pos]._thread_mutex);
        _spec_threads[pos]._thread_instructions=f;
        _spec_threads[pos]._const_args=const_args_f;
        int success=pthread_create (&_spec_threads[pos]._thread, &attr, f, const_args_f);
        if (success!=0){
                pthread_mutex_unlock(&_spec_threads[pos]._thread_mutex);
                _spec_threads.resize(pos);
                pthread_mutex_unlock(&_global_access_log_mutex);
                pthread_mutex_unlock(&_spec_threads_mutex);
                return (void*) _null_data; //the thread could not be created
        }
        _thread_index[_spec_threads[pos]._thread]=pos;
        pthread_mutex_unlock(&_spec_threads[pos]._thread_mutex);
```

```
                pthread_mutex_unlock(&_global_access_log_mutex);
                pthread_mutex_unlock(&_spec_threads_mutex);


                for (int i=0; i<pos; i++){
                        do {
                        } while (!_spec_threads[i]._commit);
                }
                bool commit_made=false;
                void* retval, *prevval;
                bool first_time=true;
                do{
                        if (!first_time){
                                prevval=retval;
                        }
                        else {
                                first_time=false;
                        }
                        success=pthread_join (_spec_threads[pos]._thread, &retval);
                        if (success!=0){
                                if (_spec_threads[pos]._commit){
                                        commit_made=true;
                                        retval=prevval;
                                }
                                else{
                                        success=0;
                                }
                        }
                } while (success==0);

                pthread_mutex_lock(&_spec_threads_mutex);
                pthread_mutex_lock(&_global_access_log_mutex);
                if (pos==(static_cast<int>(_spec_threads.size())-1)){ //if it was the last thread then it can reset the speculator.
                        _reset_speculator();
                }
                pthread_mutex_unlock(&_global_access_log_mutex);
                pthread_mutex_unlock(&_spec_threads_mutex);


                return retval;
        };

};
```