NOPL & Compiler to NOP C++ namespace multi-threading oriented solution: studies of capabilities for the x86-64 architecture

Guilherme H. K. Martini, M. Sc.
Graduate Program in Electrical and Computer Engineering (CPGEI) - Student
Federal University of Technology – Paraná
Curitiba, Brazil
ghk.martini@gmail.com

Adriano Ronszczka, M. Sc.
Graduate Program in Electrical and Computer Engineering (CPGEI) - Student
Federal University of Technology – Paraná
Curitiba, Brazil
ronszczka@gmail.com

Jean Marcelo Simão, Dr.
Graduate Program in Electrical and Computer Engineering (CPGEI) - Professor
Federal University of Technology – Paraná
Curitiba, Brazil
jeansimao@utfpr.edu.br

João Alberto Fabro, Dr.
Graduate School of Applied Computing (PPGCA) - Professor
Federal University of Technology – Paraná
Curitiba, Brazil
fabro@utfpr.edu.br

***Abstract*—This short paper presents the first study approach about the capabilities of the NOP language to support multi-threading on x86-64 computers. The initial research shows that multi-threading can provide better performance results on such architecture but a deeper development process must be done in order to reach commercial and industrial quality standards.**

***Keywords*—computing, multi-threading, computing efficiency, notification-oriented paradigm.**

I. INTRODUCTION

Modern computing deployment drives the need for the creation of new programming languages that would simplify and accelerate development of software. Doing more with less without losing control of what is being coded is one of the key reasons for continuous improvement of programming languages., This efficiency gain can mean either coding less and faster to solve a specific problem or it can mean that the same hardware architecture can become capable of doing more tasks simply by using a better suited programming paradigm [1].

The notification-oriented paradigm is a fairly new way of coding software systems that aims for better computer efficiency [2]. It has a naturally distributed system where methods, attributes and comparisons are stand-alone entities that notify each other, hence reducing the amount of wasted processing time used to poll unchanged data [2]. These characteristics make it a good and natural fit for architectures such as FPGA chips and manycore GPUs, but standard multi-core CPUs aren't an optimal target for it. Since the actual market consists of many x86 and x86-64 processors, as depicted on figure 1; and most of the high-performance modern computers in the world use this architecture, shown in figure 2, an approach for a multi-threading capability is proposed.

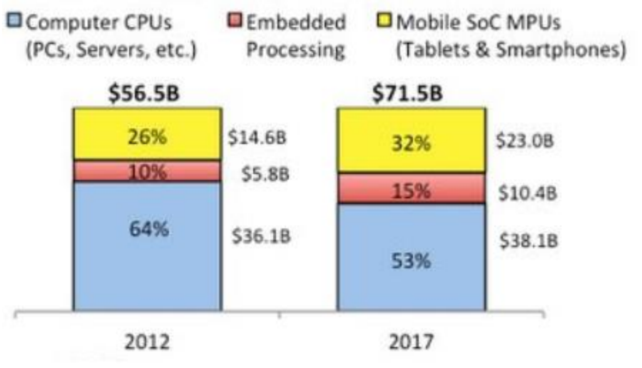


Fig. 1. Market share of x86-64 computers shift over time (light-blue), in dollars. Source: IC insights [3]

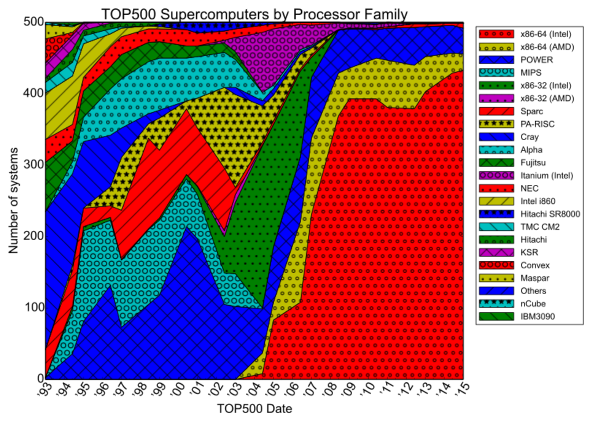


Fig. 2. Processor architectures of today's top 500 computers. Around 90% of them are x86-64. Source: Top500 [4]

II. BACKGROUND DEVELOPMENT AND OBJECTVES

By the beginning of this study, the NOP language already had a compiler that could interpret the code and transform it into a C++ code which resembles a finite state machine that uses namespaces to encapsulate its entities. Then this C++ code could be compiled and targeted to any processor architecture. This compiler version was the foundation for this work, changing it to automatically generate multi-threaded code from any NOP code became one of the goals.

A new approach for a compiler was under development at the same time this study was being made, which was the development of a mid-interpretation layer called Graf-PON. Its main intent is to generate a more flexible, intuitive and easy to use structure to target different applications, like NOP-to-C# and NOP-to-Java outputs. Any new application could benefit from characteristics of these already established programming languages in order to increase applicability of the NOP paradigm.

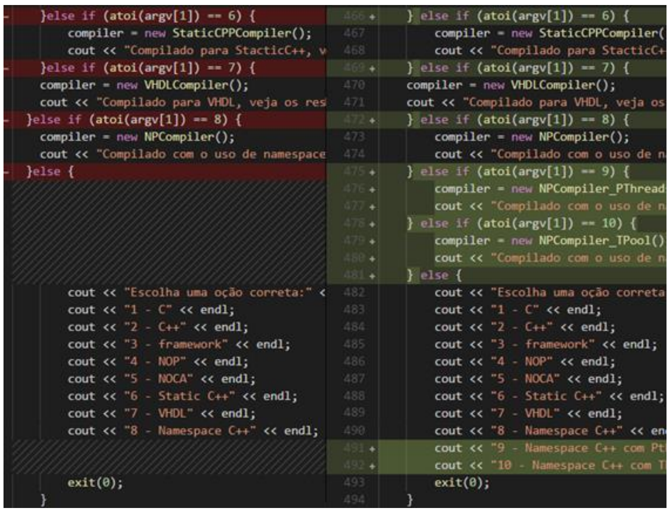
As this new approach was still being worked on, it was decided to narrow the scope only to the first compiler version that generates NOP-to-C++-namespaces code. This could then be extended in the future to the new compiler as soon as it gets finished. So, the main goals of the work were decided:

- To generate multi-threaded code from the NOP compiler C++-namespaces version.
- To compare the efficiency with the single-thread code which is already available in the first version of the compiler
- To extend multi-threading to a more complex approach of thread management, like thread pooling.

III.

III. CHANGING THE COMPILER

The compiler project already had a defined structure. It consisted of a lexical layer that was coded using Flex, and then a syntax layer that used Bison as its main tool. The entry point for changes is a virtual class that is characterized according to each type of output that is wanted from the compiler. This class had two extra characterizations for its polymorphic state: one for a PThreads approach and another for a Thread Pooling approach [5], as it is shown in figure 3.



```
}else if (atoi(argv[1]) == 6) {
    compiler = new StaticCPPCompiler();
    cout << "Compilado para StacticC++, v
}else if (atoi(argv[1]) == 7) {
compiler = new VHDLCompiler();
cout << "Compilado para VHDL, veja os res
}else if (atoi(argv[1]) == 8) {
compiler = new NPCompiler();
cout << "Compilado com o uso de namespace
}else {
    cout << "Escolha uma oção correta:" <
    cout << "1 - C" << endl;
    cout << "2 - C++" << endl;
    cout << "3 - framework" << endl;
    cout << "4 - NOP" << endl;
    cout << "5 - NOCA" << endl;
    cout << "6 - Static C++" << endl;
    cout << "7 - VHDL" << endl;
    cout << "8 - Namespace C++" << endl;
    exit(0);
}
```

```
} else if (atoi(argv[1]) == 6) {
    compiler = new StaticCPPCompiler(
    cout << "Compilado para StacticC+
} else if (atoi(argv[1]) == 7) {
compiler = new VHDLCompiler();
cout << "Compilado para VHDL, veja os
} else if (atoi(argv[1]) == 8) {
    compiler = new NPCompiler();
    cout << "Compilado com o uso de n
} else if (atoi(argv[1]) == 9) {
    compiler = new NPCompiler_PThread
    cout << "Compilado com o uso de n
} else if (atoi(argv[1]) == 10) {
    compiler = new NPCompiler_TPool()
    cout << "Compilado com o uso de n
} else {
    cout << "Escolha uma oção correta
    cout << "1 - C" << endl;
    cout << "2 - C++" << endl;
    cout << "3 - framework" << endl;
    cout << "4 - NOP" << endl;
    cout << "5 - NOCA" << endl;
    cout << "6 - Static C++" << endl;
    cout << "7 - VHDL" << endl;
    cout << "8 - Namespace C++" << en
    cout << "9 - Namespace C++ com Pt
    cout << "10 - Namespace C++ com T
    exit(0);
}
```

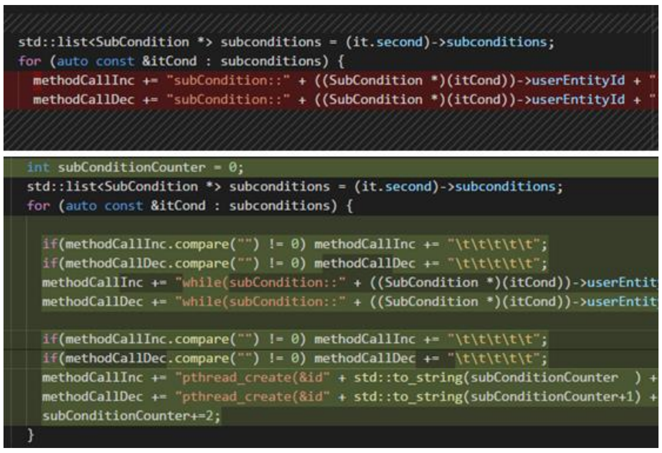
Fig. 3. Changes on the entry-point of the NOP compiler.

Inside that class, everything was copied from the C++-namespaces version and then modified accordingly. The NOP entities that received multi-threading capabilities were the methods. The reason for choosing the methods entities is simple: A big sized PON application would retain most of its processing time in the methods, not in attributes nor in conditions.

Figure 4 and 5 show where the multi-threading point was added on both approaches. It can be noted that the PThreads version always creates a thread for the work with the statement "pthread_create" along with a semaphore statement "while" right before a thread creation while the Thread Pooling version just pushes the work for a worker thread with the statement "thpool_add_work". For both pictures the code differential is done vertically, code above the white separator is the old one, followed by the replaced code.

The semaphore for the Pthreads version is needed so that a new thread is only created after the work from the previously opened thread finishes for that particular method, that is, for the same method call, the system would wait for the first iteration to finish before starting the next one. This was done to ensure data consistency and is explained in the upcoming sections.

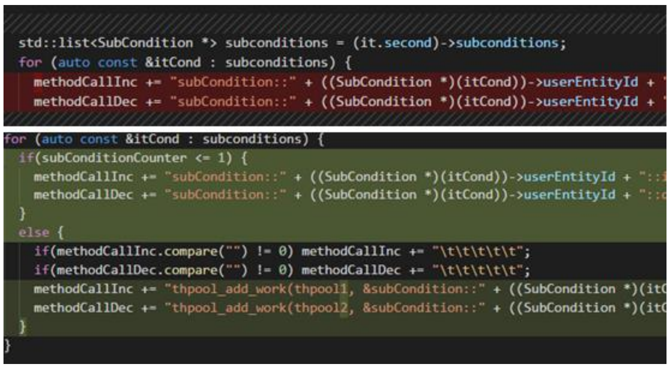
As for the Thread Pooling version, the worker threads already queue the work, so this semaphore isn't necessary, although, the main thread needs to have the information of when all worker threads are finished with the work that was queued. Not checking for this might make the program finish before the processing of all methods take place.



```
std::list<SubCondition *> subconditions = (it.second)->subconditions;
for (auto const &itCond : subconditions) {
  methodCallInc += "subCondition::" + ((SubCondition *)(itCond))->userEntityId + "
  methodCallDec += "subCondition::" + ((SubCondition *)(itCond))->userEntityId + "
```

```
int subConditionCounter = 0;
std::list<SubCondition *> subconditions = (it.second)->subconditions;
for (auto const &itCond : subconditions) {

    if(methodCallInc.compare("") != 0) methodCallInc += "\t\t\t\t\t";
    if(methodCallDec.compare("") != 0) methodCallDec += "\t\t\t\t\t";
    methodCallInc += "while(subCondition::" + ((SubCondition *)(itCond))->userEntit
    methodCallDec += "while(subCondition::" + ((SubCondition *)(itCond))->userEntit

    if(methodCallInc.compare("") != 0) methodCallInc += "\t\t\t\t\t";
    if(methodCallDec.compare("") != 0) methodCallDec += "\t\t\t\t\t";
    methodCallInc += "pthread_create(&id" + std::to_string(subConditionCounter  ) +
    methodCallDec += "pthread_create(&id" + std::to_string(subConditionCounter+1) +
    subConditionCounter+=2;
}
```

Fig. 4. Compiler changes made to allow PThreads creation on the PON compiler over the C++-Namespaces version. Above the white separator is the old portion of code, below is the changed code.



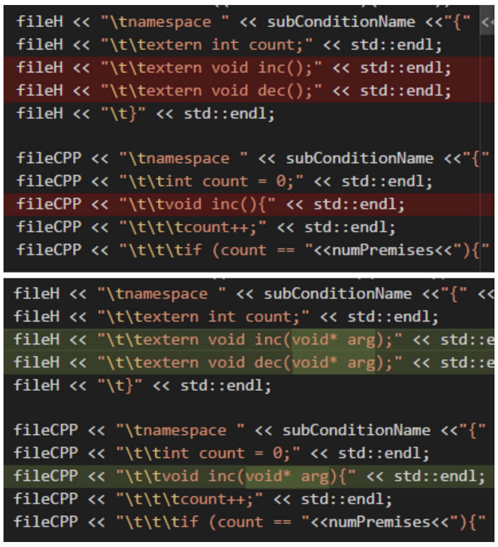
```
std::list<SubCondition *> subconditions = (it.second)->subconditions;
for (auto const &itCond : subconditions) {
  methodCallInc += "subCondition::" + ((SubCondition *)(itCond))->userEntityId + "
  methodCallDec += "subCondition::" + ((SubCondition *)(itCond))->userEntityId + "
```

```
for (auto const &itCond : subconditions) {
  if(subConditionCounter <= 1) {
    methodCallInc += "subCondition::" + ((SubCondition *)(itCond))->userEntityId + "::
    methodCallDec += "subCondition::" + ((SubCondition *)(itCond))->userEntityId + "::
  }
  else {
    if(methodCallInc.compare("") != 0) methodCallInc += "\t\t\t\t\t";
    if(methodCallDec.compare("") != 0) methodCallDec += "\t\t\t\t\t";
    methodCallInc += "thpool_add_work(thpool1, &subCondition::" + ((SubCondition *)(it
    methodCallDec += "thpool_add_work(thpool2, &subCondition::" + ((SubCondition *)(it
  }
}
```

Fig. 5. Compiler changes made to allow Thread Pooling creation on the PON compiler over the C++-Namespaces version. Above the white separator is the old portion of code, below is the changed code.

Besides the work on changing adding thread creation/pooling, some other changes were necessary, such as:

- Addition of PThread and Thread Pooling libraries.
- Changes on method call arguments in order to comply with Pthreads and Thread Pooling libraries' parameters, shown on figure 6.

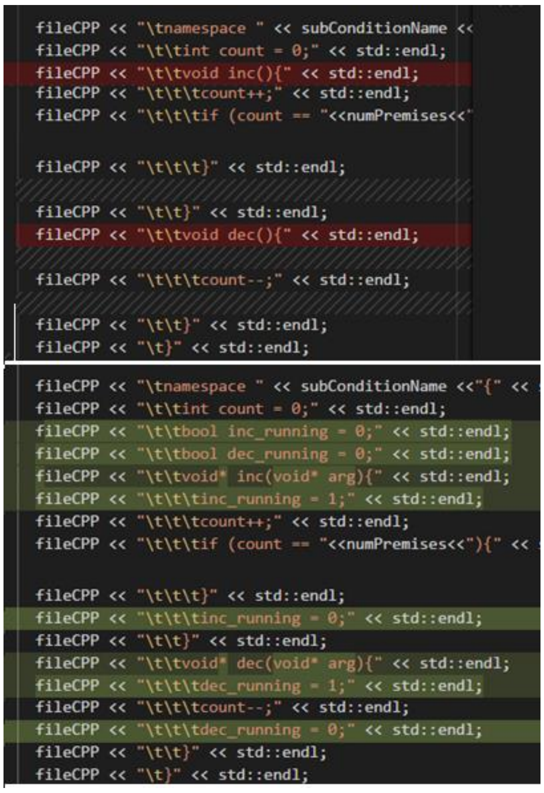


```
fileH << "\tnamespace " << subConditionName <<"{" <<
fileH << "\t\textern int count;" << std::endl;
fileH << "\t\textern void inc();" << std::endl;
fileH << "\t\textern void dec();" << std::endl;
fileH << "\t}" << std::endl;

fileCPP << "\tnamespace " << subConditionName <<"{"
fileCPP << "\t\tint count = 0;" << std::endl;
fileCPP << "\t\tvoid inc(){" << std::endl;
fileCPP << "\t\t\tcount++;" << std::endl;
fileCPP << "\t\t\tif (count == "<<numPremises<<"){"
```

```
fileH << "\tnamespace " << subConditionName <<"{" <<
fileH << "\t\textern int count;" << std::endl;
fileH << "\t\textern void inc(void* arg);" << std::e
fileH << "\t\textern void dec(void* arg);" << std::e
fileH << "\t}" << std::endl;

fileCPP << "\tnamespace " << subConditionName <<"{"
fileCPP << "\t\tint count = 0;" << std::endl;
fileCPP << "\t\tvoid inc(void* arg){" << std::endl;
fileCPP << "\t\t\tcount++;" << std::endl;
fileCPP << "\t\t\tif (count == "<<numPremises<<"){"
```

Fig. 6. Argument changes of method calls.

- Declaration and instantiation of PThread and Thread pools
- Destruction of these objects at the end of their use
- Implementation of semaphores for the PThread case, which is shown on figure 7.



```
fileCPP << "\tnamespace " << subConditionName <<
fileCPP << "\t\tint count = 0;" << std::endl;
fileCPP << "\t\tvoid inc(){" << std::endl;
fileCPP << "\t\t\tcount++;" << std::endl;
fileCPP << "\t\t\tif (count == "<<numPremises<<"

fileCPP << "\t\t\t}" << std::endl;

fileCPP << "\t\t}" << std::endl;
fileCPP << "\t\tvoid dec(){" << std::endl;

fileCPP << "\t\t\tcount--;" << std::endl;

fileCPP << "\t\t}" << std::endl;
fileCPP << "\t}" << std::endl;
```

```
fileCPP << "\tnamespace " << subConditionName <<"{" <<
fileCPP << "\t\tint count = 0;" << std::endl;
fileCPP << "\t\tbool inc_running = 0;" << std::endl;
fileCPP << "\t\tbool dec_running = 0;" << std::endl;
fileCPP << "\t\tvoid* inc(void* arg){" << std::endl;
fileCPP << "\t\t\tinc_running = 1;" << std::endl;
fileCPP << "\t\t\tcount++;" << std::endl;
fileCPP << "\t\t\tif (count == "<<numPremises<<"){" <<

fileCPP << "\t\t\t}" << std::endl;
fileCPP << "\t\t\tinc_running = 0;" << std::endl;
fileCPP << "\t\t}" << std::endl;
fileCPP << "\t\tvoid* dec(void* arg){" << std::endl;
fileCPP << "\t\t\tdec_running = 1;" << std::endl;
fileCPP << "\t\t\tcount--;" << std::endl;
fileCPP << "\t\t\tdec_running = 0;" << std::endl;
fileCPP << "\t\t}" << std::endl;
fileCPP << "\t}" << std::endl;
```

Fig. 7. Semaphore implementation for the PThread mode.

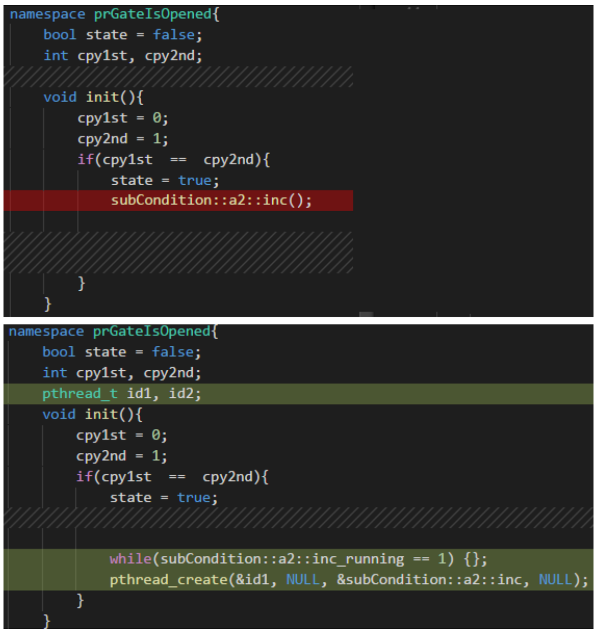
It is worth mentioning that this was an iterative process in which a lot of testing was done along the way in order to get to the optimal output code, which is further analyzed in the next section.

IV. Output Code Analysis

After the changes on the compiler were made, an analysis on the output C++ code was necessary. As one of the goals is automatic generation, a NOP code that is compiled has to generate working code as before, but with multi-threading capabilities.

The NOP code used as input is the Electronic Gate project, which already is available as an example project inside the compiler source code. It was used to validate the compiler changes during its development.

Figure 8 shows how a PThread call was implemented and how the wait for the semaphore gets coded in the target C++ code. This can be further improved in the future as the while loop is making the main execution thread wait sequentially for all work to be finished. Instead, a pool of information could be queried and more worked could be pushed sequentially until completion of the execution. As the example NOP code is relatively simple, it was concluded that for a first performance comparison, the generated code is good "as is".

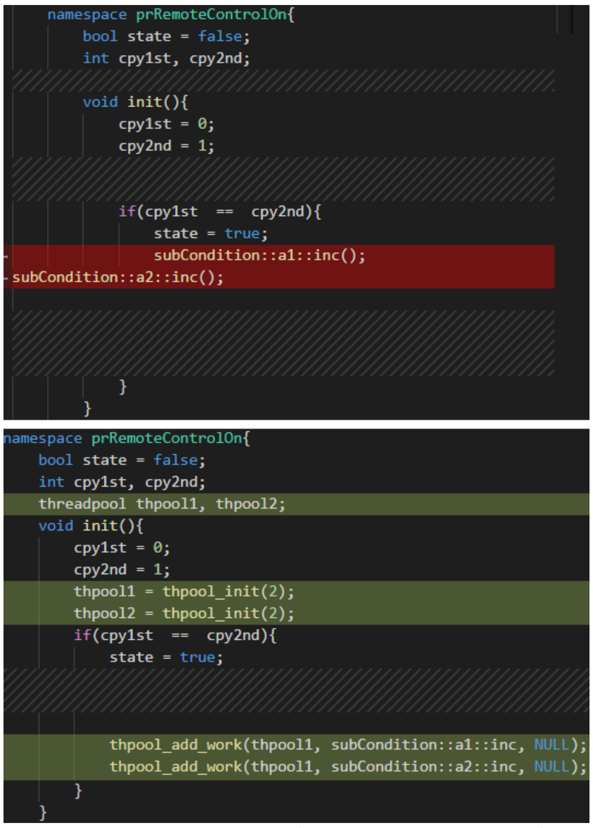


```
namespace prGateIsOpened{
    bool state = false;
    int cpy1st, cpy2nd;

    void init(){
        cpy1st = 0;
        cpy2nd = 1;
        if(cpy1st  ==  cpy2nd){
            state = true;
            subCondition::a2::inc();


        }
    }
```

```
namespace prGateIsOpened{
    bool state = false;
    int cpy1st, cpy2nd;
    pthread_t id1, id2;
    void init(){
        cpy1st = 0;
        cpy2nd = 1;
        if(cpy1st  ==  cpy2nd){
            state = true;


            while(subCondition::a2::inc_running == 1) {};
            pthread_create(&id1, NULL, &subCondition::a2::inc, NULL);
        }
    }
```

Fig. 8. PThread code result after compiling a NOP code of the Electronic Gate project.

Figure 9 is the equivalent result for the Thread Pooling version, which instead of opening and closing a thread for every function call, the calls are just queued in a worker thread.



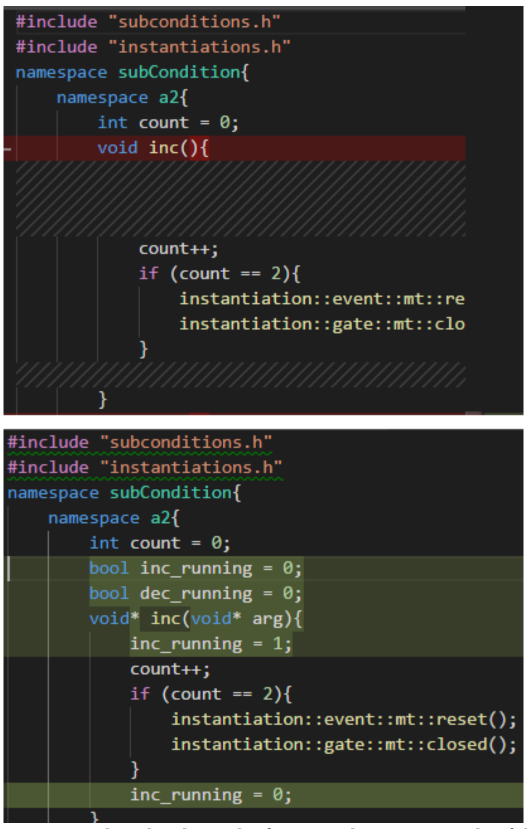
```
        namespace prRemoteControlOn{
            bool state = false;
            int cpy1st, cpy2nd;

            void init(){
                cpy1st = 0;
                cpy2nd = 1;


                if(cpy1st  ==  cpy2nd){
                    state = true;
                    subCondition::a1::inc();
subCondition::a2::inc();


                }
            }
```

```
namespace prRemoteControlOn{
    bool state = false;
    int cpy1st, cpy2nd;
    threadpool thpool1, thpool2;
    void init(){
        cpy1st = 0;
        cpy2nd = 1;
        thpool1 = thpool_init(2);
        thpool2 = thpool_init(2);
        if(cpy1st  ==  cpy2nd){
            state = true;


            thpool_add_work(thpool1, subCondition::a1::inc, NULL);
            thpool_add_work(thpool1, subCondition::a2::inc, NULL);
        }
    }
```

Fig. 9. Thread Pooling code result after compiling a NOP code of the Electronic Gate project.

Also, as explained in the compiler code, more changes can be seen in the generated code such as library inclusions, object creation/destruction and a waiting verification for work completion on the Thread Pooling version.

One last noticeable change is the semaphore on every method for the Pthreads version, depicted on figure 10.



```
#include "subconditions.h"
#include "instantiations.h"
namespace subCondition{
    namespace a2{
        int count = 0;
        void inc(){

            count++;
            if (count == 2){
                instantiation::event::mt::re
                instantiation::gate::mt::clo
            }

        }
```

```
#include "subconditions.h"
#include "instantiations.h"
namespace subCondition{
    namespace a2{
        int count = 0;
        bool inc_running = 0;
        bool dec_running = 0;
        void* inc(void* arg){
            inc_running = 1;
            count++;
            if (count == 2){
                instantiation::event::mt::reset();
                instantiation::gate::mt::closed();
            }
            inc_running = 0;
        }
```

Fig. 10. PThread code result after compiling a NOP code of the Electronic Gate project.

After compilation of the C++ code using GCC and verification that the final application was running as it should, a performance analysis was made.

V. Performance Comparisson

The performance check between the original time spent to run the application versus the PThread version and the Thread Pooling version was made by a simple clock counter. Whenever the code execution started, the clock counter starts. When the application is finished, the clock count is summed up and translated to an estimated time that was taken by the computer to run the application. All runs were made under the same computer, with the same operational system and with the same task priority on its scheduler.

As the Electronic gate project is a simple program, a CRC32 calculation was added to every method that is called in order to increase processing burden. To further increase that burden and to evaluate how more costly programs would run, a “for” loop was added to the CRC32 calculation, so a tendency can be verified as burden is changed.

Many runs for the same amount of CRC32 loops were made in order to confirm that timing was consistent between them. For a same simulation, times varied less than 1% in all attempts.

Figure 11 shows a graph with the results. As processing burden increases, Thread Pooling tends to provide better results than PThread version, which is better than the original one.

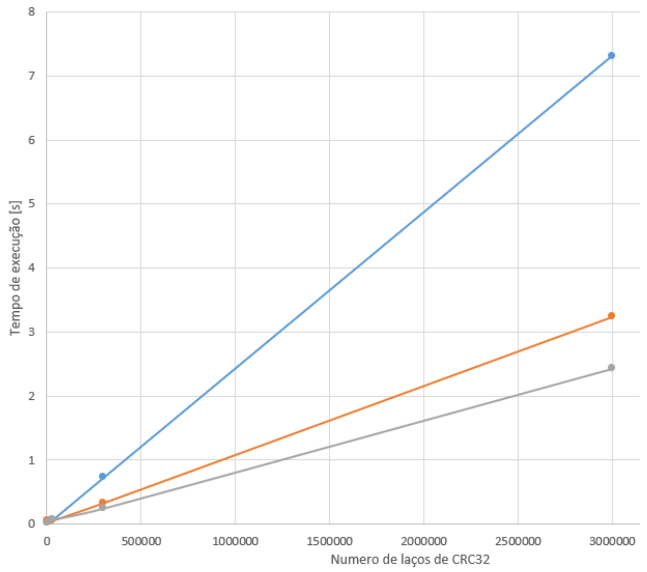


Fig. 11. Time taken to run the Electronic Gate project after it was compiled with the original version (blue), PThread version (orange) and Thread Pooling version (gray).

VI. Conclusion

With the gathered data it is concluded that all goals for this work were achieved, the compiler now generates codes with multi-threading resources, more than one approach was tested and their results were compared. It was noted that whenever the program gets more complex, a better way to handle attribute changes from concurrent methods is needed in order to grant data consistency. Also, in case methods from past events are left for future processing after some sooner event triggers a method right away a time discrepancy will happen; to solve this, a broker can be implemented, being this a suggestion for future work.

References

[1] Rojas, Raúl; et al. *Plankalkül: The First High-Level Programming Language and its Implementation*. Institut frame Informatik, Freie Universität Berlin, Technical Report B-3/2000, February, 2000. Avaliable: ftp://ftp.mi.fu-berlin.de/pub/reports/TR-B-00-03.pdf

[2] Banaszewski, Roni; et al. *Paradigma Orientado a Notificações: Avanços e Comparações*. UTPFR, 2009.

[3] *The McClean report [Online]*. IC Insights, 2018.
Avaliable:http://www.icinsights.com/services/mcclean-report/report-contents/

[4] *Statistics | TOP500 Supercomputer Sites [Online]*. Top500, 2014.
Avaliable: https://www.top500.org/statistics/

[5] Seferidis, Johan Hanseen. *C-Thread-Pool [Online]*, April, 2017.
Available: https://github.com/Pithikos/C-Thread-Pool