Comprehension of Thread Scheduling for the C++ Programming Language

Attila Gyén
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
gyenattila@gmail.com

Norbert Pataki
Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
patakino@elte.hu

2021 International Conference on Data and Software Engineering (ICoDSE) | 978-1-6654-9453-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICoDSE53690.2021.9648489

***Abstract*—Writing parallel programs – whether it is a multi-threaded or multi-processed – can often be a daunting task. Developers need to pay close attention to a lot of things related with threads and processes. It is possible that one of these things will be ignored against their will. If this happens the program easily can go into a faulty state. In many cases, this means all that is left is to terminate the program manually. There may also be cases where a poorly written parallel program, despite the calculations run on multiple cores or on multiple computers such as social network servers for example, the program itself proves to be slower than its sequential counterpart. This paper investigates to find out whether there is an "actual" parallelism between threads or processes or where there are cases when one of them has to wait for the others to finish their execution. This getting to the point that the program could have been solved with a much simpler sequential program. We propose an approach and its implementation that results in the comprehension of thread schedule. We deal with C++ threads therefore we do not mind the different hardware elements, the language constructs provide abstraction over CPUs and operating systems. This approach is based on static analysis and its implementation takes advantage of the Clang compiler infrastructure.**

***Index Terms*—C++, code comprehension, multi-threading, static analysis**

I. Introduction

Due to the increasing computing capacity, the demands on software are also increasing. As a result, often it is no longer sufficient for a program to simply "just" function well, it has become an expectation to do this in the most efficient way. Increasing efficiency can be approached from several points:

1) *Expanding the hardware* – in this case, the physical boundaries should be considered that can be a limitation [1]. For larger companies, it is common for an expansion of a server farm to involve building an entirely new server farm. New hardware integration involves experts and will continue to be required to perform maintenance in the future.
2) *The exploitation of hardware* – this is more common in hardware programming or when the resources are limited and the hardware expansion is not an option. The goal is to improve the program by forcing it to do calculations faster. This can be done by optimizing the source code of the program. During the code optimization, the developers do not have to pay so much attention to a hardware limitations. Their goal is to boost up their program to run faster as much as possibly while maintaining consistency. However, it has its own boundaries.
3) *Choose another programming language* – there are cases when the software uses a programming language that no longer meets the needs of it. An obsolete programming language does not provide the opportunities, or at least not a required level to develop a more optimal, faster program. In this case, a new programming language should be considered. Choose this option in well-justified cases.
4) *Upgrade the current programming language to a newer version* – for most programming languages, a newer standard appears from time to time with more powerful language implementations and features. It is possible that the older version of the programming language has not yet implemented the required language constructs. The upgrade process is not as easy as it looks like. In the newer version, there could be functions which behave differently and because of this the program that has worked well so far can produce undefined behavior or even worse, crash. Larger companies have separate teams to decide whether a new standard of a programming language is ready-to-use with the current program or not.

Unfortunately, it seems that these approaches are not always possible or not always the best choices. It can take a lot of time, energy and money to implement new hardware or rewrite working software from scratch in a completely different programming language, not to mention the usage of new language elements that may differ from those which were used in previous versions, thereby causing unknown errors and unpredictable behavior. Because of these reasons, there is one more possible solution:

5) *Parallelization* – purely sequential programs no longer stand alone or only in certain cases, due to performance degradation [2]. This solution requires more attention from developers [3]. Unlike the first and second solutions, there are no physical boundaries, and it is not necessary to build the system in a different programming

language. Even so, there may be major changes, but they are not always necessary.

However, dealing with parallel execution is quite difficult. The formation of source code is the mold of sequential execution. Moreover, parallelization can be the root cause of many runtime problems (interferences, deadlocks, etc.), therefore software maintenance of multithreaded code requires many experiences. Code comprehension tools may mitigate these difficulties. Straightforward solutions are required for supporting sustainable development goals.

Programmers are eager for tools to mitigate the maintenance of multithreaded applications. In this paper, we propose an approach which can help in this mitigation in case of multi-threaded C++ application. The C++ standard does not have any notion of a multi-processing therefore in the next sections we will talk about multi-threaded and async future techniques. Our approach is based on static analysis, therefore our approach takes advantage of the source code. No execution is required, thus the tool can be deterministic and fast. Our solution uses the Clang compiler infrastructure to detect the relationship of the execution threads which one is started or finished. These solutions definitely support the sustainable development.

The rest of this paper is organized as follows. In Section II, the related work is discussed. In Section III, we provide an overview of static analysis, code comprehension and the applied tools. We propose our approach in Section IV and present our tool in Section V. Finally, this paper concludes in Section VI.

II. Related work

The analysis of parallel programs has been addressed by several researchers in the past. Different approaches have been proposed for different programming languages.

Many details of the execution and runtime behavior of parallel programs are analysed [4]. However, the proposed approach is based on a dynamic analysis technique as opposed to static analysis. The topic of data races related to thread errors is discussed [5]. In this technique, a detector is presented that can be used to place dynamic annotations in the program and the analysis uses these annotations to examine the program. It also handles these errors dynamically at runtime, although there is a static equivalent to this method [6]. SyncTrace's [7] approach to multi-threaded programs had a completely different perspective, because it evaluates the chronological order of blocking operations, like finding the use of synchronization and I/O operations, and the relationships between threads. Reference [8] focuses on how to schedule parallel tasks in a single-threaded systems to minimize their cost. A method to detect shared variables of multithreaded C++ code is proposed [9]. This approach uses the Clang compiler and static code analysis to determine problematic parts of the code. Reference [10] that discusses the process of thread security analysis, also with the help of the Clang, but focuses on the part of multi-threaded programming that provides the program various annotation declarations to enforce multi-threaded programming guidelines in C and C++ languages.

III. Static analysis

Static analysis is a method in which the source code is processed without its execution [11]. This approach is similar to the usual behavior of the compilers, an abstract syntax tree (AST) is built. However, static analyzers do not deal with low-level code generation. An essential feature of static analyzers is the detection of bugs in the early phase of development. The typical tools find undiscovered bugs in the code or checking conventions [11]. They can help in the validation of subtle compatibility issues. With the help of static analyzers, one can measure software metrics. Refactoring, obsfucating or rejuvenating code also requires these approaches.

Code comprehension is an essential approach in the modern software engineering. Code comprehension tools enable to locate specific code snippets, for instance, jump to the definition of a function or a class [12]. For this purposes, code comprehension takes advantage of static analysis in order to increase the productivity of software developers and make easier the software maintenance. However, code comprehension supports multithreaded programming in a limited way [9].

The purpose of code comprehension tools is to make the understanding of the source code easier, which is not as simple as it sounds at first hearing [13]. In general, there are two aspects to achieve this. First approach is *What* and second is *How*. The best way to understand a code written by another person is to breaking it into smaller parts – objects and classes – and see their properties. Their running methods, relations and communication between them. So at the end of the day it can be said that code comprehension can be used to discover the connection between different parts of code.

Clang is an open source compiler for C, C++, and Objective-C. It is built on the top of LLVM compiler infrastructure. Many software developer companies support the Clang project, for instance, Apple and Google. Clang's modular architecture makes it popular and enables to develop further build tools [11].

IV. The proposed approach

Modern programming languages allow developers to write parallel or concurrent programs [14]. Most of them enable to create multiple threads and multiple standalone processes. These techniques are also known as multi-threading or multi-processing. There are cases, when the previously mentioned techniques are not available in a language, like in JavaScript, but the opportunity is still there to use a so called async-await blocks, which is not a real multi-threading nor multi-processing, but allows the code to not block the main thread. The purpose of these methods is improving the run-time of the program and run the application smoother.

The fork-join model is typical for the parallelization. Modern solutions use this technique. In this model, when working with multi-threaded or multi-processed programs, the program starts as a traditional sequential program. However, by calling a special function, the interpreter, or the compiler – depends on programming language – marks another entry point, then instructs the processor to start executing these threads or

processes as well. Parallel programs, either multi-threaded and multi-processed can have a lot of setbacks. Examples include improper use of shared memory space, starvation, or deadlock [15]. Different operating systems do not handle threads and processes equally at the machine code level [16]. Because of this fact it can happen that a program runs effectively on one operating system, but runs much slower on another system.

The fact that a program runs on multiple threads or processes does not necessarily means that it will be faster or more optimal than its sequential counterpart. It is possible that a thread can only start its execution if the other threads before have already finished their own job. As a result, there will still be a quasi-sequential execution, despite the fact that the execution is done on multiple threads. Moreover, since creating each thread requires extra steps on the part of the processor. It has some overhead [17] – largely operating system dependent. For this reason, the program may be even slower that if it was written simply sequentially. To decide whether to use a multi-threaded or multi-process program requires considering exactly what problem we want to solve. There is no clear winner, the solution always depends on the problem.

This paper aims at the establishment of a static code analysis method that helps in the comprehension of multi-threaded C++ programs. We implemented this approach with the utilization of the Clang compiler infrastructure.

Consider the following program:

```
#include <thread>

void foo() { /* do something */ }

int main()
{
    auto t1 = std::thread(foo);
    t1.join();
    auto t2 = std::thread(foo);
    t2.join();
}
```

The execution of the program shown above will be sequential even tough it runs on multiple threads. This is due to the fact that `t2` thread is not constructed until the execution of `t1` thread is finished and attached back to the main thread. That is because of the `t1.join();` statement which specifies that do not continue main thread's execution until `t1` is finished, as it is shown in the Fig. 1.

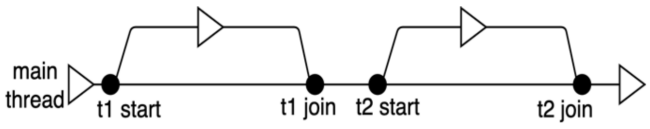

Fig. 1. Sequential execution of multi-threaded program

Make a few changes to the program as follows:

```
#include <thread>

void foo() { /* do something */ }

int main()
{
    auto t1 = std::thread(foo);
    auto t2 = std::thread(foo);
    t1.join();
    t2.join();
}
```

After the changes, the program will actually run in parallel. While `t1` working on separate thread, `t2` can be constructed on the main thread and start its own execution, as it is shown in Fig. 2.

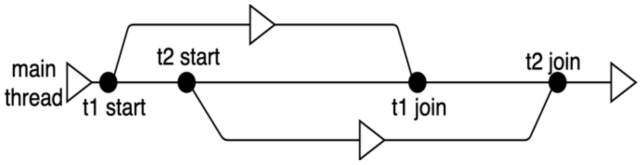

Fig. 2. Parallel execution of multi-threaded program

During the development, the main idea of our tool was to determine whether threads are started and joined within a program in a way that affects the start of other threads while also affecting the execution time of the program. Put simply, to avoid the case shown in Fig 1. It should be mentioned that this tool does not deal with the implementation of the in-thread program and any other problems with threads mentioned earlier. Just focuses on being able to determine whether a program is executed sequentially - or not - despite the fact that operations are performed on multiple threads.

Static code analyzers, compilers and code comprehension tools are not aware of how multi-threaded programs work. They just detect whether there is a new thread instantiation in the source code and when to start it. For this reason, we wanted this tool to be able to serve as a kind of code comprehension support tool to not just detect threads and processes, but also when they are started and joined back to their calling thread. Doing all of this statically. It will be able to help developers as they debug their program by displaying various warning messages [18] that may point to the problematic parts of the code [19]. We put the focus on the C++ programming language as it was mentioned earlier, more precisely on the standard thread and future classes published in the C++11 standard. During the research, the last officially adopted C++17 standard and the Clang compiler was used.

It is necessary to mention that C++ does not define a separate keyword to start a thread. At the moment of instantiation, the execution of the new thread is started. The language distinguishes two essential types of thread management techniques:

1) `std::thread` - represents a single thread of execution. The `join()` method must be used to re-attach threads back to the calling thread;

2) `std::future` - it provides an asynchronous operation. There are two options for executing the called function in the future object. First one with the help of the `async` function template. In this case, it will invoke the function in a clearly new thread. Second option is with the help of the `deferred` policy. It will invoke the function in the calling thread. To receive information from future objects the `get()` function must be called.

When calling the `join()` or the `get()` methods, the program waits until the execution is finished inside the thread. It is necessary to know where a new thread is created within the program and when it is joined.

This information can be accessed using the Clang compiler's built-in switch that can be used to access the Abstract Syntax Tree (*AST*) from the source code. Then the relevant pieces of information can be filtered out from it, such as the type of the variables that have been instantiated or their location within the code. The AST built by Clang differs from the abstract syntax tree built by other compilers in that it is very similar to the written C++ code itself. One example of this is that parentheses and compile time constants are available in non-reduced form. Then the AST can be reduced, because the original tree can be thousands or hundreds of thousands of lines long due to the use of the included header files. To understand it will require proper knowledge of the tokens and keywords it uses, at least those that carry relevant information.

1) `VarDecl` - indicates a variable declaration and contains information about the variable itself, such as the name and type, or whether the variable was used at any point in the program. The latter is indicated by keyword *used* in the AST;
2) `DeclRefExpr` - stores all information about how the declaration is referenced in a particular term.

The meaning of the following constructs `std::__1::thread`, `std::__1::future`, `async`, `deferred`, `join`, and the `get` have already been discussed previously. The tokens and keywords mentioned above are shown in Fig. 3, 4 and 5. The significant tokens are framed.

```
<col:12, col:70> col:8 used f 'std::__1::future<void>':'std::__1::future<void>' cinit
<col:12, col:70> 'future<typename __invoke_of<typename decay<void (&)()>>>'

<col:12, col:70> 'future<typename __invoke_of<typename decay<void (&)()>::type>>'
<col:12, col:17> 'future<typename __invoke_of<typename decay<void (&)()>>>'

<col:12, col:17> 'future<typename __invoke_of<typename decay<void (&)()>::type>>'
'future<typename __invoke_of<typename decay<void (&)()>::type>::type>'

<col:23, col:57> 'std::__1::launch' adl
<col:42> 'std::1::launch (*)(std::__1::launch, std::__1::launch)'
<col:42> 'std::__1::launch (std::__1::launch, std::__1::launch)' lvalue
<col:23, col:36> 'std::__1::launch' EnumConstant 0x7fa6fde648 'async'
<col:44, col:57> 'std::__1::launch' EnumConstant 0x7fa6fde648 'deferred'
```

Fig. 3. future, async and deferred tokens from AST dump

As it seems, it really has a lots of information stored about the source code. Once the reduction is done, the next step is to search for the variable declarations. To do this, the lines that contain the `VarDecl` token should be processed. A distinction

```
`-VarDecl 0x7ff7b01a79d0 <col:3, col:27> col:8 used t 'std::__1::thread'
  `-ExprWithCleanups 0x7ff7b01a8aa0 <col:12, col:27> 'std::thread'
    `-CXXFunctionalCastExpr 0x7ff7b01a89b0 <col:12, col:27> 'std::thread'
      `-CXXBindTemporaryExpr 0x7ff7b91a89b0 <col:12, col:27> 'std::thread'
        `-CXXConstructExpr 0x7ff7b01a8978 <col:12, col:27> 'std::thread'
          `-DeclRefExpr 0x7ff7b01a7ab8 <col:24> 'void ()' lvalue
```

Fig. 4. VarDecl, thread and DeclRefExpr tokens from AST dump

```
0x7ff7b01a8af0 <col:3, col:5> '<bound member function type>' .get() 0x7ff7b0141830
0x7ff7b01a8ad0 <col:3> 'std::__1::future<void>':'std::__1::future<void>' lvalue
0x7ff7b01a8b90 <line:11:3, col:10> 'void'
0x7ff7b01a8b60 <col:3, col:5> '<bound member function type>' .join 0x7ff7b005be70
0x7ff7b01a8b40 <col:3> 'std::__1::thread':'std::__1::thread' lvalue Var 0x7ff7b01be19
```

Fig. 5. get and join tokens from AST dump

must be made between variables of the thread or future type and all other types. When the separation is done in order to reduce the final output of the tool a constraint can be places on future-type variables. As it was mentioned earlier, if a future type thread is instantiated in deferred mode the task it perform will not be executed in a new thread, but in the calling one. For this reason only variables of future type needs to be processed if they are instantiated in asynchronous mode. Fig. 6 presents the high-level workflow of our approach that we implemented.

V. The Proposed Tool

Recently, our implementation is a command line interface (CLI) tool that processes C++ source code. The name of the file to be scanned must be specified as an input, therefore on one hand, it is easy to use. On the other hand, the visualization is premature yet. Our future work includes the integration of the tool and an Integrated Development Environment (IDE), as well.

Consider the following program:

```
#include <thread>
#include <future>

void foo() { /* do something */ }

int main()
{
    auto t = std::thread(foo);
    auto f =
      std::async(std::launch::async, foo);
    t.join();
    f.get();
}
```

This program will actually run in parallel, as neither thread will have to wait others to finish executing. However, if modify the program as follows:

```
#include <thread>
#include <future>

void foo() { /* do something */ }
```

```
int main()
{
    auto f =
      std::async(std::launch::async, foo);
    f.get();
    auto t = std::thread(foo);
    t.join();
}
```

The result of the first thread is awaited by the program before the execution of the second thread can start, thus the execution of the program will be quasi-sequential. In this case, the tool will generate the following output:

```
[{
    "threadName": "f",
    "threadType": "std::future",
    "startedInRow": 7,
    "joinedInRow": 8,
    "canHoldUp": true
 },
 {
    "threadName": "t",
    "threadType": "std::thread",
    "startedInRow": 9,
    "joinedInRow": 10,
    "canHoldUp": false
}]
```

The final output of the tool is a JavaScript Object Notation (*JSON*) object. It uses colloquial field names, like `threadName` and `threadType` to indicate the name and the type of the thread. The `startedInRow` and `joinedInRow` fields store information about the row in which the thread started to execute and the row in which the `join()` or the `get()` function was executed. A field `canHoldUp` stores information about whether a thread could cause a problem that could potentially increase the run-time of the program. For the last thread in the program, this field will always get a false value because it will no longer affect the start of other threads.

False positive cases are often problematic when static analysis tools are involved. However, our tool is nice enough because threads only in unavailable, dead code may cause false positives. Our future work includes handling this scenario. Moreover, expressive visualization may improve the code comprehension, and even more sophisticated predictions for cases where the execution of a thread is conditional or the thread has been detached in the meantime from the calling thread, therefore it is an essential future work, as well.

VI. Conclusion

Code comprehension tools can be really helpful in many cases. There are tools and different techniques to measure the performance of programs. These can be used to determine how long certain functions of a program run, but they are not able to decide whether the functions run in parallel or not. They only focus on the execution time and how much of the resources are used during run time. Other tools can deal with threads or processes on some level, but as we can see there are cases for which they are not prepared. In most cases, we need to execute the analysed program for these specific pieces of information.

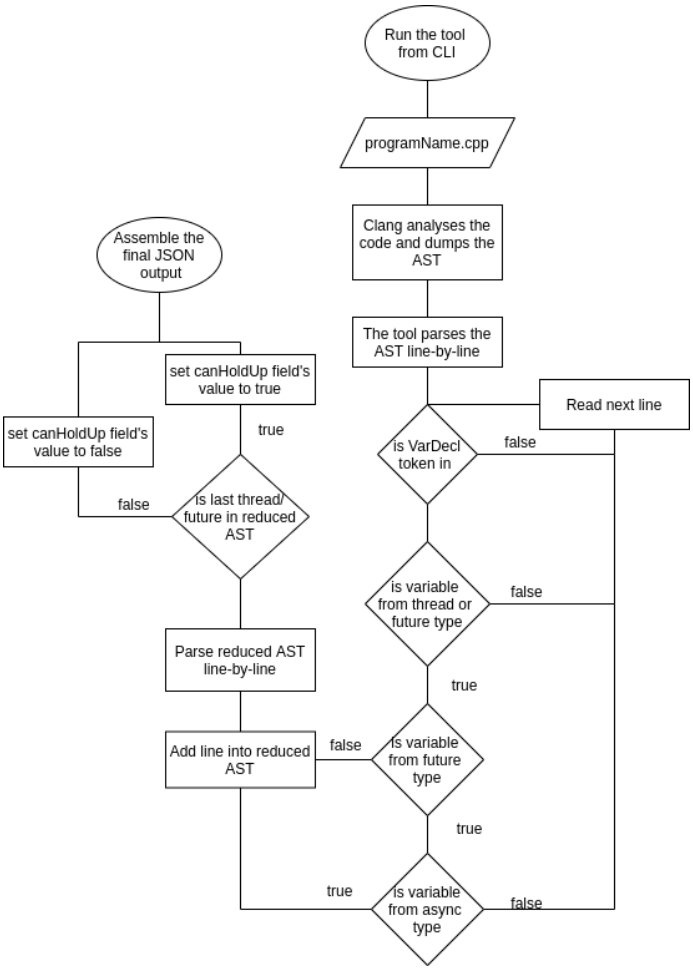

Fig. 6. Algorithm of our tool operation

Testing complex programs is a time-consuming task. There is a need to re-run the program almost after every change and also restart the test environment. Our main motivation was to find out where the previously mentioned problems with threads can happen without executing the main program.

We defined an approach to realize scheduling of threads in C++. This method is based on the static analysis, we took advantage of the source code and its AST counterpart. We implemented this approach with the help of the Clang compiler infrastructure.

During the development session of our tool one of the basic concept was to make it use as easy as possible. To this end, we have made every effort to ensure that the user does not have to bother using the tool. We tried our best to optimize each step as much as possible. Due to this, the tool is able to perform its task without any user intervention from running the program to be analyzed to producing the final output. Each

step was thought through and tested several times until the final solution was completed. With the help of this tool, the user could receive reports in advance of the possible danger before executing their application. We also payed a lot of attention on that the final output of this tool is displayed as eloquent and concisely as possible, comfortable for the human eye to read and suit many programming languages, as well.

A further goal is that in the future is to be able to recognize not only `std::thread` and `std::future` type threads but also those from third-party libraries. It will be expanded to other programming languages, like Java or Python, and more accurate and formatted warning messages will be displayed and it will have more eloquent user interface. It was a significant achievement to correctly define the analysis of the AST. It took a long time to identify each major token and keyword and figure out how to best analyze the variables within the program and then separate the ones that are important from the rest.

REFERENCES

[1] R. Marciniak, "Role of new IT solutions in the future of shared service model," Pollack Periodica, vol. 8, no 1, pp. 187–194, 2013.

[2] A. Lindholm and S. Nenonen, "A conceptual framework of CREM performance measurement tools," Journal of Corporate Real Estate, vol. 8, no. 3, 2006, pp. 108–119.

[3] F. Magoulès and C. Venet, "Parallel domain decomposition methods for beam-tracing," Pollack Periodica, vol. 7, no. 2, pp. 3–23, 2012.

[4] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization", in Proceedings of the ACM Conference on Computer and Communications Security, Chicago, USA, October, 4-8, 2010, 2010, pp. 133–142.

[5] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer – data race detection in practice," in Proceedings of the Workshop on Binary Instrumentation and Applications, New York, USA, December 12, 2009, 2009, pp. 62–71.

[6] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, USA, October 19-22, 2003, 2003, pp. 237–252.

[7] B. Karran, J. Trümper, and J. Döllner, "SyncTrace: Visual thread – interplay analysis," in Proceedings of the 1st Working Conference on Software Visualization, Eindhoven, The Netherlands, September, 27-28, 2013, 2013, pp. 1–10.

[8] R. Khogali and O. Das, "Cost minimization for scheduling parallel single-threaded heterogeneous speed-scalable processors," in Parallel and Distributed Systems (ICPADS) 2013 International Conference, Seoul, Korea, December 15–18, 2013, 2013, pp. 265–274.

[9] A. Gyén and N. Pataki, "Discovering shared variables for comprehension of multithreaded C++ Programs," in Collection of Abstracts, 13th Joint Conference on Mathematics and Computer Science, online, October 1–3, 2020, 2020, pp. 76–77.

[10] D. Hutchins, A. Ballman, and D. Sutherland, "C/C++ thread safety analysis," in IEEE 14th International Working Conference on Source Code Analysis and Manipulation, Victoria, BC, Canada, September 28-29, 2014, 2014, pp. 41–46.

[11] B. Babati, G. Horváth, V. Májer, and N. Pataki, "Static analysis toolset with Clang," in Proceedings of the 10th International Conference on Applied Informatics, Eger, Hungary, January 30–February 1, 2017, 2017, pp. 23–29.

[12] Z. Porkoláb and T. Brunner, "The codecompass comprehension framework," in Proceedings of the 26th Conference on Program Comprehension (ICPC'18), Gothenburg, Sweden, May 28–29, 2018, 2018, pp. 393–396.

[13] C. Bennett, D. Myers, M.-A. Storey, and D. German, "Working with 'monster' traces: Building a scalable, usable sequence viewer," in Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis, Vancouver, BC, Canada, October, 28-31, 2007, 2007, pages 1–5.

[14] E. W. Burton and D. J. Simpson, "Space efficient execution of deterministic parallel programs," IEEE Transactions on Software Engineering, vol. 25, no. 6, pp. 870–882, 1999.

[15] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," Transactions on Computers, vol. 38, no. 2, pp. 49–60, 1989.

[16] F. A. Bower, D. J. Sorin, amd L. P. Cox, "The impact of dynamically heterogeneous multicore processors on thread scheduling," in IEEE Micro, June 27, 2008, 2008, pp. 17–25.

[17] R. Feldmann, P. Mysliwietz, and B. Monien, "Studying overheads in massively parallel rain/max-tree evaluation," in Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures, Cape May, USA, June, 27–29, 1994, 1994, pp. 94–103.

[18] J. Berthold and R. Loogen, "Visualizing parallel functional program runs: Case studies with the eden trace viewer," in Proceedings of the International Conference Parallel Computing, Jülich/Aachen, Germany, September, 4-7, 2007, 2007, pp. 121–128.

[19] M. Bedy, S. Carr, X. Huang, and C.-K. Shene, "A visualization system for multithreaded programming," Special Interest Group Computer Science Education Bulletin, vol. 32, no 1, pp. 1–5, 2000.