Tasks Execution in Multithreaded Program According to the Dependency Graph

Kostiantyn Nesterenko [1,*, †], Inna Stetsenko and Eduard Zharikov [1,†]

[1] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 37 Prospect Beresteiskyi, Kyiv, 03056, Ukraine

Abstract

Performance is one of the main non-functional requirements for software. As a result of the increase in the number of cores in central processing units in recent decades [1], the use of multithreading technology has become a primary means of improving software performance. This study analyzes the problems that arise from developing multithreaded programs and ways to address them. A method for managing the execution of tasks in a multithreaded program based on a given dependency graph is proposed and its implementation in the C++ language is demonstrated. Its aim is to reduce the resource intensity of software development and increase its reliability by addressing problems typical of developing multithreaded programs. The results of experimental research on a test set of tasks are provided, demonstrating increased performance through the use of the proposed method.

Keywords

Software, multithreading, parallel computing, performance, reliability, dependency graph

1. Introduction

The primary goal of using a multithreaded approach during software development is to improve the per formance indicators of the program code. However, another equally important aspect of multithreaded development is maintaining software reliability.

Using multithreading and synchronization tools can lead to a number of errors: Data race, Race Condition, Deadlock, Livelock, and Starvation [2]. The presence of these problems in the code during program execution can lead to various consequences, from crashes to unpredictable or incorrect pro gram behavior. Approaches to solving multithreaded issues can vary significantly depending on the programming language, framework, development tools, etc. However, all of them have certain draw backs in their application, as they can negatively impact both the resource intensity of the development process and the performance of the developed program code. These approaches can be fundamentally divided into two types: those that prevent the occurrence of errors and those that allow the engineer to detect errors when they occur.

2. Overview of existing solutions

2.1. Data race

A Data race occurs when two threads simultaneously access the same memory area, with at least one of them performing a write operation [2]. A classical mechanism to prevent Data race in multithreaded code is the use of synchronization. By synchronizing data access, it is possible to

14th International Scientific and Practical Conference from Programming UkrPROG'2024, May 14-15, 2024, Kyiv, Ukraine

* Corresponding author.

† These authors contributed equally.

k.nesterenko@kpi.ua (K. Nesterenko); stiv.inna@gmail.com (I. Stetsenko); eduard.zharikov-fiot@lll.kpi.ua (E. Zharikov)

0000-0003-3921-4324 (K. Nesterenko); 0000-0002-4601-0058 (I. Stetsenko); 0000-0003-1811-9336 (E. Zharikov)

ensure that a specific memory area can only be modified by one thread at a time or that it will not be modified during its reading by a thread. Synchronization tools are an integral part of modern programming languages and continue to evolve, expanding developers' capabilities in managing multithreaded programs and their resources, such as the C++ Concurrency Library [3].

However, the use of synchronization primitives has its drawbacks, the main one being a reduction in program code performance. Besides the fact that the application of a primitive, such as acquiring a mutex by a thread, takes additional time to execute, a significant amount of time can be lost by threads waiting for a resource protected by a synchronization primitive to be released.

To improve the performance of program code, developers increasingly use approaches to software development without blocking synchronization primitives [4]. However, this significantly increases the risk of Data race problems. Various tools are used to detect this problem directly during code execution, such as the Data Race Detector in the Go programming language [5] or by statically analyzing the written program code [6]. Recently, there have been developments related to using artificial intelligence to detect Data races in program code [7].

However, using such tools to detect Data races also has drawbacks, as it requires the introduction of additional tools in the software development and testing process, increasing the overall resource intensity of development. Furthermore, after detecting an error, it must be processed and corrected by the developer, and the program tool must be rechecked for new errors.

2.2. Race condition

Unlike Data race, Race condition is a much more challenging problem to detect because its nature is not technical but semantic. The definition of the term Race condition has long been a subject of discussion, but currently, it can be formalized as a problem arising from the order of operations execution in threads affecting the program's result [8].

Similar to the Data race error, there are certain tools to detect Race condition that warn the developer of the potential occurrence of this problem through static code analysis [9] or by identifying dangerous patterns during code execution [10]. However, the accuracy of such detection is often unsatisfactory. These methods are also prone to false positives, requiring the developer to spend additional time and resources verifying the tool's data.

There are no formalized methods and tools to avoid Race condition in the general case. Reducing the number of such errors or avoiding them can only be achieved by using certain methods and architectural approaches to multithreaded software development [11].

2.3. Deadlock

Deadlock is a state in a multithreaded program where two or more threads cannot continue their execution due to mutual blocking. The classic strategy to avoid Deadlock is a method where a thread acquires all the resources it needs exclusively at the beginning of its execution [12]. This prevents the formation of cyclical dependencies between two threads. The thread either executes or waits until all the resources it needs are released. However, this approach negatively impacts performance, as, in the general case, the thread does not need all the resources at each moment of execution, and thus, they could be used by other threads.

There are many different methods and algorithms for detecting Deadlock. For example, by using timers, it is possible to control the execution time of certain critical sections of code in threads and, if this execution time exceeds a certain threshold, notify the developer of a potential Deadlock [12]. It is also worth noting that there are Deadlock recovery strategies [13], which are extremely important for systems with increased reliability requirements. However, it is evident that recovery strategies negatively affect software performance.

2.4. Starvation

Starvation is a state in a multithreaded program where one of the threads cannot perform a task due to the inability to access the necessary resources. This state can arise due to design errors in multithreaded programs when resources protected by synchronization primitives are used by threads for a long time. As a result, other threads spend significant time waiting for the necessary resource to be released. To avoid Starvation, prioritization algorithms [14] or task scheduling [15] are usually used, guaran teeing that the task will be performed at a certain moment and allowing tasks to be distributed in such a way as to minimize time spent waiting for resources to be freed.

Various software tools can be used to detect Starvation, examining metrics about the execution time of individual functions in threads and allowing the developer to identify and fix problematic sections of the program code [16]. Such tools are often integrated into software testing automation processes to constantly monitor the program code state and notify developers of anomalies or performance issues.

In the result of the analysis of existing problems of multithreading and possible solutions, it can be stated that solving these problems significantly increases the resource intensity of multithreaded software development. Additionally, some of the existing solutions negatively affect the performance of the software, which puts the developer in a difficult position of choosing between performance and reliability of the software product.

Therefore, the problem of reducing the resource intensity of multithreaded software development without compromising its reliability and performance requires further research.

3. Method for managing the execution of tasks in a multithreaded program based on a given dependency graph

3.1. General description of the method

The main idea of the method for managing the execution of tasks in a multithreaded program based on a dependency graph is to define the process of executing tasks, which the computations are divided into, as a dependency graph where each vertex corresponds to a specific task, and the graph edges denote dependencies between these tasks. Tasks can be executed in parallel in threads, considering the constraints imposed by the dependencies between them.

The method for managing the execution of tasks in a multithreaded program based on a dependency graph allows describing the execution process of multithreaded program code in a clear and well structured form and avoids or detects problems arising from the use of task execution synchronization.

3.2. Formalized description of the method

Suppose that for successful program execution, it must complete K tasks that can be described by the set $A = \{A_0, \ldots, A_{K-1}\}$. If the program's result depends on the order of execution of tasks A_i and A_j, and only one of the possible results is correct, it means the tasks are in a semantic dependency. Let Y be a directed acyclic graph with vertices $G = \{G_0, \ldots, G_{K-1}\}$ and arcs U. Each task A_j from the set A corresponds to the vertex G_j of the graph Y. If there is a semantic dependency between tasks A_i and A_j, or if tasks A_i and A_j use a shared resource during their execution, then there is a connection between the corresponding vertices G_i and G_j in the graph, and the arc (G_i, G_j) is assigned to it in the graph. In this way, the set of arcs is created:

$$U = \{(G_i, G_j) | i, j \in \mathrm{N}, i \neq j, 0 \leq i < K, 0 \leq j < K\}. \quad (1)$$

Tasks can be dynamically loaded during program execution, creating new vertices and arcs in the graph, and deleted upon successful completion. Thus, each task has a representation in the dependency graph from creation to completion. Since the start of the task, the correspondent vertex

is not available for creating a new arc. When a new task is created, the added arcs should not form a cycle in a graph. If all the added arcs are directed to the new vertex (G_i, G_x) then the new graph will be acyclic. Indeed, if no path can be built through the new vertex then a new cycle cannot be built in the graph. Similarly, if all added arcs are directed from the new vertex (Gx, Gj) then the new graph will be acyclic. However, if among added arcs are both cases, to and from the new vertex, then the following condition should be checked. For each pair of added arcs (G_i, G_x) and (G_x, G_j), a path from G_j to G_i should not be found in graph U:

$$\nexists P: G_j \longrightarrow \cdots \longrightarrow G_i\,, \tag{2}$$

where P is a path in graph Y, $\longrightarrow$ is an existing directed connection between vertices in graph.

Indeed, if the path exists, then the following cycle would be found in a new graph:

$$G_i \longrightarrow G_x \longrightarrow G_j \longrightarrow \cdots \longrightarrow G_i\,. \tag{3}$$

Task A_x can be executed in the program if and only if there is no arc (G_i, G_x) in the dependency graph. If task Ax is executed, it is deleted from the graph along with the arcs (G_x, G_j):

$$\{(G_i, G_x) | i \in \mathrm{N}, i \neq x, 0 \leq i < K\} \cup \{(G_x, G_j) | j \in \mathrm{N}, x \neq j, 0 \leq j < K\} = \emptyset \implies G := G \setminus G_x\,. \tag{4}$$

Program execution is considered successful when the set of vertices in the dependency graph Y is empty at the end of the program, meaning that all tasks loaded for execution have been completed: $G = \emptyset, U = \emptyset$.

3.3. Method based software

The basis for implementing the method of task execution management in a multithreaded program based on a dependency graph is the implementation of the GraphManager class (Figure 1), which will be responsible for the order of task execution. Accordingly, this class must control the list of available tasks, select tasks that can be executed, and update the existing dependencies after their execution. To represent a task as a vertex in the dependency graph, it is necessary to create a GraphNode class, which will contain information about the task to be executed and a list of tasks it depends on and tasks that depend on it. Dependencies between tasks are established by the developer after the corresponding GraphNode class objects have been added to the GraphManager. During the implementation of the method, the developer must represent the types of tasks available in the program as separate SomeConcreteJob classes, which are descendants of the abstract AbstractJob class.

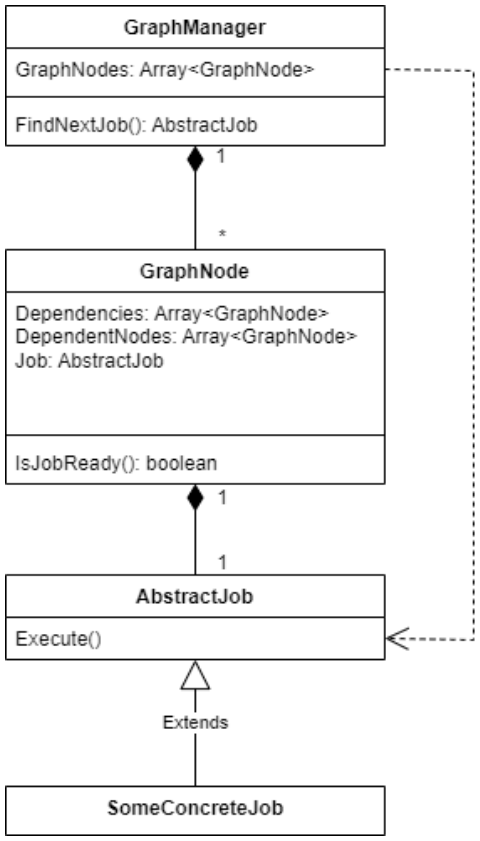

Figure 1: UML diagram of method implementation.

3.4. Resolving concurrency issues using a method

To avoid the Race condition problem, developers need only specify dependencies between tasks, ensuring the correct sequencing necessary for program output integrity. This method guarantees that tasks with established dependencies will execute in the specified order relative to each other.

To address Data race issues and improve performance by reducing the need for synchronization primitives to protect specific objects, developers can establish dependencies among tasks requiring exclusive access to certain resources while allowing parallel execution of tasks that do not modify shared resources.

While the proposed method does not eliminate Deadlock and Livelock issues entirely, it simplifies implementing mechanisms to detect these issues prior to executing the dependency graph. Dependency graph edges can denote not only semantic dependencies between tasks but also dependencies between tasks requiring exclusive access to shared resources. Deadlock problems arise when synchronized resources are mutually dependent between two threads. With this method, detecting potential Deadlocks reduces to finding cycles in the graph, a problem for which many algorithms exist [17]. Since cycle detection need not occur before every program execution but only after changes to the dependency graph, applying a depth-first search-based cycle detection algorithm suffices for this method.

Similarly, this approach applies to Livelock issues. Absence of cycles in the graph and construction of dependencies among tasks needing exclusive access to shared resources prevent such issues during program execution. The proposed method does not entirely prevent Starvation problems. However, it minimizes them and facilitates implementing runtime monitoring mechanisms without relying on external control mechanisms. Task execution management based on the dependency graph allows developers to configure the graph so that no task waits for the release of a specific shared resource after starting its execution. Instead, such a task simply will not

start until the resource becomes available, adhering to task dependencies. Consequently, system resources can be directed toward executing tasks currently without dependencies or those whose dependencies have already been fulfilled.

Nevertheless, this method does not prevent scenarios where a task depends on many other tasks, thereby preventing them from executing until it completes successfully. Detecting such issues involves implementing mechanisms to monitor the graph's state, recording execution times, waiting times, and other relevant data for each task. With this information, developers can modify the code to address problematic areas and improve the performance of the software tool.

4. Experimental investigation of the efficiency of task execution management method in a multithreaded program based on a specified dependency graph

4.1. Description of the test software

To evaluate the effectiveness of the proposed method, a program was developed in C++ that, using the Strategy design pattern, allows executing the same set of tasks either through a conventional implementation of a multithreaded program or based on a dependency graph. Thread management utilizes the Thread Pool design pattern (Figure 2).

To enable configuration of which approach to use for task execution, the IJobManager interface (Figure 3) was created. This interface encapsulates the mechanism through which the Thread Pool selects the next task for execution.

```
#pragma once
#include <vector>
#include <thread>

class IJobManager;

class ThreadPool
{
public:
    ThreadPool(IJobManager* pJobManager,
        size_t numThreads = std::thread::hardware_concurrency());
    ~ThreadPool();

    void Start();
    void Stop();

    void ThreadMainLoop();

    IJobManager* GetJobManager() { return m_jobManager; }

private:
    size_t m_numThreads { 0 };

    IJobManager* m_jobManager { nullptr };
    std::vector<std::thread> m_threads;
};
```

Figure 2: ThreadPool template implementation.

```cpp
#pragma once

#include<memory>

class BaseJob;

class IJobManager
{
public:
    virtual void AddJob(BaseJob* pJob) = 0;
    virtual std::unique_ptr<BaseJob> GetNextJob() = 0;

    virtual void OnJobFinished(int jobId) = 0;

    virtual bool IsFinished() = 0;
};
```

Figure 3: Class IjobManager.

The IJobManager interface is implemented in the DefaultJobManager class (Figure 4) and the GraphJob Manager class (Figure 5). The DefaultJobManager class is implemented using a traditional queue based on the "first in, first out" (FIFO) principle. Accordingly, it reflects the classic scenario of writing a multithreaded program where synchronization responsibility lies entirely on synchronization primitives.

```cpp
#pragma once
#include "IJobManager.h"
#include <mutex>
#include <queue>

class DefaultJobManager
    : public IJobManager
{
public:
    virtual void AddJob(BaseJob* pJob) override;
    virtual std::unique_ptr<BaseJob> GetNextJob() override;

    virtual void OnJobFinished(int jobId) override {}

    virtual bool IsFinished() override;

private:
    mutable std::mutex m_jobQueueMutex;
    std::queue<std::unique_ptr<BaseJob>> m_jobQueue;
};
```

Figure 4: Class DefaultJobManager.

```cpp
#pragma once
#include "IJobManager.h"
#include <mutex>
#include <queue>
#include <map>

#include "GraphNode.h"

class BaseJob;

class GraphJobManager :
    public IJobManager
{
public:
    virtual void AddJob(BaseJob* pJob) override;
    virtual std::unique_ptr<BaseJob> GetNextJob() override;

    virtual void OnJobFinished(int jobId) override;

    virtual bool IsFinished() override;

    void AddJobDependency(int jobId, int dependentJobId);

private:
    int m_nextJobId { 0 };

    mutable std::mutex m_jobQueueMutex;
    std::queue<std::unique_ptr<BaseJob>> m_jobQueue;

    std::map<int, std::unique_ptr<GraphNode>> m_graphMap;
};
```

Figure 5: Class GraphJobManager.

The GraphJobManager, in turn, is implemented using the proposed method. Within the class, there is a data structure called m_graphMap, which associates each task with a corresponding node in the graph, described using the GraphNode class (Figure 6).

```cpp
#pragma once
#include <memory>
#include <vector>
#include <atomic>

class BaseJob;

class GraphNode
{
public:
    GraphNode(BaseJob* job);

    void AddDependency(GraphNode* node);
    void ResolveDependencies();

    void IncUnresolvedDependencies();
    void DecUnresolvedDependencies();

    bool CanExecuteJob();
    std::unique_ptr<BaseJob> ExtractJob();

private:
    std::unique_ptr<BaseJob> m_job;
    std::atomic_int m_unresolvedDependencies { 0 };
    std::vector<GraphNode*> m_dependentNodes;

    bool m_bJobExtracted { false };
};
```

Figure 6: Class GraphNode.

Each instance of the GraphNode class internally stores the number of unresolved dependencies for its corresponding task, or in other words, the number of edges entering this vertex in the graph. Additionally, each instance holds pointers to other instances of the GraphNode class, effectively describing edges leaving this vertex in the graph. When all dependencies for a graph vertex are satisfied, it is moved to the execution queue in the GraphJobManager class. The GraphJobManager class also includes the AddJobDependency function, which allows specifying dependencies between tasks, effectively creating a new edge in the dependency graph.

4.2. Description of the test data and testing scenarios

To test the Thread Pool, a pool of 4 threads was configured. The test set of tasks consists of 4 groups of 100 tasks each, which can be represented as sets $A = \{A_0, \ldots, A_{99}\}$, $B = \{B_0, \ldots, B_{99}\}$, $C = \{C_0, \ldots, C_{99}\}$, $D = \{D_0, \ldots, D_{99}\}$. Within each group, simulating real-world conditions for a multithreaded task, it is assumed that there is a specific resource for which each task obtains exclusive access using a mutex during its execution. After obtaining access, the task waits for 10 ms, simulating computation using the Busy wait approach [18], and successfully completes its execution. The use of Busy wait specifically avoids the impact of thread optimization mechanisms implemented in the operating system on the obtained result.

To obtain more accurate results for comparison, the test set of tasks is executed 100 times for each implementation. The following metrics are collected: fastest execution time, slowest execution time, and average execution time.

To measure the execution time of the program for computing the specified set of tasks using the dependency graph constructed based on the proposed method, the GraphJobManager will be used instead of the DefaultJobManager. When tasks are added in GraphJobManager, a graph node

described by the GraphNode class is created for each task. Then, using the AddJobDependency function, dependencies between tasks are established, effectively adding edges in the dependency graph. In this case, the dependency object is the resource allocated for each group of tasks. Thus, using the AddJobDependency function, dependencies between tasks will be created as follows: each task Ai depends on the execution of task Ai-1, Bi depends on Bi-1, Ci depends on Ci-1, and Di depends on Di-1 where i ∈ [1, 99].

It is worth noting that since the classical approach to executing multithreaded tasks is implemented using a queue, the total task execution time depends on their order in this queue. Accordingly, three testing scenarios were developed for the classical approach: best case, worst case, and realistic.

The best-case scenario considers tasks in the queue arranged such that the time spent waiting for access to resources by tasks is minimized. This scenario corresponds to arranging tasks in the queue where each belongs to a different group and requires a different resource for execution. Thus, the sequence of tasks in the queue will look like: $\{A_0, B_0, C_0, D_0, \ldots, A_{99}, B_{99}, C_{99}, D_{99}\}$.

The worst-case scenario considers tasks in the queue arranged such that the time spent waiting for access to resources by tasks is maximized. This scenario corresponds to arranging tasks in the queue where groups of tasks are sequentially placed in the queue. Thus, the sequence of tasks in the queue will look like: $\{A_0, \ldots, A_{99}, B_0, \ldots, B_{99}, C_0, \ldots, C_{99}, D_0, \ldots, D_{99}\}$. The realistic scenario considers tasks randomly placed in the queue, simulating conditions closer to those that may occur during real program execution.

Testing results

The experimental results represented in Table 1 demonstrate that compared to the realistic and worst case execution scenarios, the proposed method showed significantly higher performance. This result was achieved because a proposed method allows developers to specify the task execution order by introducing dependencies between them, thereby reducing the impact of synchronization mechanisms on the overall program execution time.

Table 1
Results of experimental performance evaluation of task execution under various processing scenarios

Task execution scenario	Execution time, ms		
	Fastest	Slowest	Average
Best case	1003.42	1008.47	1004.67
Worst case	3915.63	3919.09	3917.58
Realistic	1453.54	1643.43	1542.71
Dependency graph	1009.00	1013.64	1010.74

The experimental results of the best-case, realistic-case, and the case of dependency graph scenarios are represented in detail in Figure 7. The performance evaluation was done on 100 executions of the program.

In the best-case execution scenario, the results show minimal performance difference in favor of the classical approach. This can be easily explained by the fact that in this case, the task execution order is identical in both approaches, but the proposed method incurs additional time for graph construction and update. However, considering that such a scenario may rarely occur in real-world conditions.

Therefore, according to the results of the experimental study, it has been proven that the proposed method allows to increase the performance of the test program on average by approximately 35%, compared to the classical approach: (1542.71 – 1010.74) / 1542.71 ·100% = 34.48%.

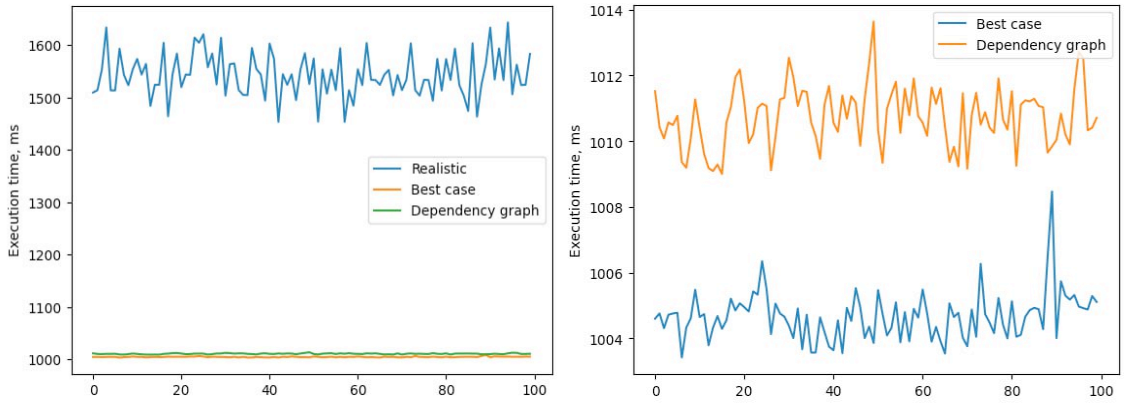

Figure 7: Experimental performance evaluation of task execution.

5. Conclusion

An analysis of the challenges arising during the development of multithreaded programs was con ducted. For each issue, current methods of resolution were discussed, outlining their advantages and disadvantages.

A method for managing task execution in a multithreaded program based on a specified dependency graph was proposed. The general idea, formal description, and programmatic details of the method were provided, along with how this method addresses the identified issues and the advantages it provides to developers compared to existing methods.

A C++ program and a suite of tasks were developed to measure the performance of the proposed method against the classical approach of executing tasks in threads. After analyzing the obtained results, it was concluded that the proposed method improves the performance of the test program on average by 35% compared to the classical approach.

Future plans include further development of the method's use for developing multithreaded programs based on dependency graphs, aiming to create a comprehensive development environment for such programs.

References

[1] S. Borkar, A. Chien, The future of microprocessors, Commun. ACM 54(5) (2011) 67–77. doi:10.1145/1941487.1941507.

[2] Y. Lin, Multithreaded programming challenges, current practice, and languages/tools support, in: 2006 IEEE Hot Chips 18 Symposium (HCS), Stanford, CA, 2006, pp. 1–134. doi:10.1109/HOTCHIPS.2006.7477737.

[3] M. Gregoire, Multithreaded Programming with C++, in: Professional C++, Fourth Edition, 2021, pp. 915–967. doi:10.1002/ 9781119695547.ch27.

[4] K. Fraser, T. Harris, Concurrent programming without locks, ACM Trans. Comput. Syst. 25(2) (2007). doi:10.1145/1233307.1233309.

[5] M. Chabbi, M. Ramanathan, A study of real-world data races in golang, in: PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2022, pp. 474-489. doi:10.1145/3519939.3523720

[6] V. Kahlon, Y. Yang, S. Sankaranarayanan, A. Gupta, Fast and accurate static data race detection for concurrent programs, in: Lecture Notes in Computer Science, 4590 (2007) 226-239. doi:10.1007/978-3-540-73368-3_26.

[7] L. Chen, X. Ding, M. Emani, T. Vanderbruggen, P.-H. Lin, C. Liao, Data race detection using large language models, in: Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 215-223.

[8] R. Netzer, B. Miller, What are race conditions? - some issues and formalizations, ACM letters on programming languages and systems 1 (1992) 74-88. doi:10.1145/130616.130623.
[9] C. Flanagan, S. Freund, Detecting race conditions in large programs, in: PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2001, pp. 90–96. doi:10. 1145/379605.379687.
[10] M. Yousaf, M. Sindhu, M. Arif, S. Rehman, Efficient identification of race condition vulnerability in C code by abstract interpretation and value analysis, in: 2021 International Conference on Cyber Warfare and Security (ICCWS), Islamabad, Pakistan, 2021, pp. 70-75. doi:10.1109/ICCWS53234.2021. 9702954.
[11] J. Ortega-Arjona, The manager workers pattern, in: Proceedings of the 9th European Conference on Pattern Languages of Programms (EuroPLoP '2004), Irsee, Germany, July 7-11, 2004, pp. 53–64.
[12] M. Singhal, Deadlock detection in distributed systems, Computer 22 (1989) 37 – 48. doi:10.1109/ 2.43525.
[13] Y. Park, P. Scheuermann, H.-L. Tung, A distributed deadlock detection and resolution algorithm based on a hybrid wait-for graph and probe generation scheme, in: CIKM '95: Proceedings of the fourth international conference on Information and knowledge management, 1995, pp. 378–386. doi:10.1145/ 221270.221648.
[14] R. Jabbour, I. Elhajj, Saf-ps: Starvation avoidance for priority scheduling, in: 2008 5th International Multi-Conference on Systems, Signals and Devices, Amman, Jordan, 2008, pp. 1–6. doi:10. 1109/SSD.2008.4632789.
[15] A. Gawanmeh, W. Mansoor, S. Abed, D. Kablaoui, H. Faisal, Starvation avoidance task scheduling algorithm for heterogeneous computing systems, in: 2021 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 2021. doi:10.1109/CSCI54926.2021.00339.
[16] S. Abbaspour, M. Saadatmand, S. Eldh, D. Sundmark, H. Hansson, A model for systematic monitoring and debugging of starvation bugs in multicore software, in: SCTDCP 2016: Proceedings of the 1st International Workshop on Specification, Comprehension, Testing, and Debugging of Concurrent Programs, 2016, pp. 7-11. doi:10.1145/2975954. 2975958.
[17] A. Nesterenko, Cycle detection algorithms and their applications, Journal of Mathematical Sciences 182 (2012) 518-526. doi:10.1007/s10958-012-0755-x.
[18] J. Blieberger, B. Burgstaller, B. Scholz, Busy wait analysis, in: Lecture Notes in Computer Science 2655 (2003), 142–152. doi:10.1007/ 3-540-44947-7_10.