DOI 10.34658/jacs.2017.2.69-88

JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 25 No. 2 (2017), pp. 69-88

Resolving Classical Concurrency Problems Using Outlier Detection

Mateusz Smoliński[1]

[1]*Lodz University of Technology*
Faculty of Technical Physics, Information Technology and Applied Mathematics/Institute of Information Technology
ul. Wólczańska 215, 90-924 Lodz, Poland
mateusz.smolinski@p.lodz.pl

Abstract. *In this paper outlier detection is used to determine anomaly between tasks to prevent occurrence of resource conflicts in prepared schedule. Determined conflictless schedule bases on controlling access of tasks to groups of shared resources. Proposed approach allows to prepare conflictless schedule of efficient parallel task processing without resource conflicts and is dedicated to environments of task processing with high contention of shared resources. In this paper the outlier detection is used to resolve two classical concurrency problems: readers and writers and dining philosophers. In opposition to other known solutions of concurrency problems, proposed approach can be applied to solve different problems and do not require to use additional mechanisms of task synchronization. The universality of proposed approach allows to prepare conflictless schedule even in environments, where classical concurrency problems will be significantly expanded and complicated.*

Keywords: *Resource conflict as outlier, mutual exclusion, deadlock avoidance, cooperative concurrency control, adaptive conflictless scheduling.*

1. Introduction

Task processing in modern computer architectures is often performed in parallel. Taking the general task definition as sequence of instructions, which create separate execution path that process data, that definition corresponds to many task processing environments. For example, tasks can be determined as single process thread, tasklet or transaction. If parallel processed tasks use disjoint resource sets, there is no need to synchronize them. In the situation where two tasks executed in parallel require to use the same instance of shared resource and at least one of them perform write operation on this resource then conflict occurs [1, 2]. Then program developer encounter difficulties, because he has to take into account all dependencies between tasks and respectively includes them in controlling access to shared resources. The software developer is responsible for the selection of proper scheduling algorithm and synchronization techniques to the specific concurrency problem. Also protection from task starvation or uneven allocation of shared resources is the demanding job. Additionally in task processing environments, many occurrences of resource conflict can cause the tasks deadlock, which is the worst situation in task processing, because all tasks participating in deadlock have no progress in execution. Along with the increasing the number of resource dependencies between tasks the difficulty in development of correct program increases.

The developer can use adaptive conflictless scheduling, which is an alternative to existing synchronization techniques (i.e. Two Phase Locking or Hierarchy of Resources [3, 4]) used in known solutions of concurrency problem to provide parallel tasks processing without resource conflicts. The task execution plan determined by adaptive conflictless scheduling guarantees that no resource conflicts occur between tasks processed in parallel. The software developer, which uses adaptive conflictless scheduling is only obligated to determine each task boundaries with all required by this task shared resources.

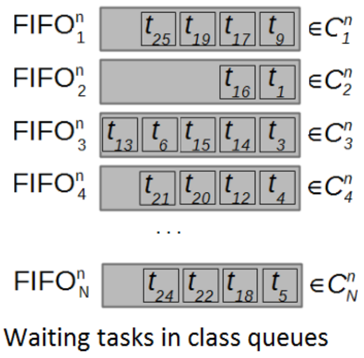
The concept of conflictless scheduling bases on binary representation of tasks resources. The concept of rapid anomaly detection between tasks allows to prepare the conflictless schedule for the specific state of environment. The global conflictless schedule is adaptive, because it is composed as a sequence of conflictless schedule determined to the specific environment state. Therefore the effective detection of resource conflicts in successive environment states is required to prepare adaptive conflictless scheduling. To prepare adaptive conflictless schedule a dedicated model of task resources representation and additional data structures like task classes or conflict array are required [5, 6]. This approach was extended by using association rules [7]. Adaptive conflictless scheduling can be applied only in task processing environments that meets all criteria presented in next section.

1.1. Outliers detection in high contention environment

The resource contention occurs when multiple tasks executed in parallel attempt to use at least one of shared resources. This means that not all resource contentions generate resource conflict. The resource conflict do not occur in contention situation, when all operations requested by tasks on shared resource do not change its state. Otherwise, the resource conflict occurs during the contention. In task processing environment where tasks at the same time read and write the same instance of resource, the higher level of contention causes more resource conflicts. The conflictless task scheduling is recommended to use in task processing environments with high level of contention and limited number of reusable, shared resources, which state can be changed by tasks.

Assuming that the task execution time does not have to be known a priori and shared resource states are changed as a result of not coordinated task executions in high contention environment, then the resource conflict is a random event. This event can be considered as a contextual outlier, which base on attributes like time and resources representation for tasks. Therefore effective outlier detection allows to prepare schedule, that assures task executions in parallel without resource conflicts. The conflictless scheduling approach requires additional assumptions for the task processing environment, that are presented in the next section. The outlier detection has many applications, this allows to determine anomaly or novelty in large data sets [**?**, 8, 9, 10]. There is many techniques of outlier detection, but for detection of resource conflicts in the high contention environment will be used the dedicated binary model, which includes representation of dependencies between tasks and shared resources. The usage for task the binary representation of all its reuired shared resources provides rapid outlier detection.

Each task has dedicated representation of all its shared resources, which can be assigned statically (i.e. by programmer) or dynamically (i.e. by software manager). For each task the global resources representation includes two binary resources identifiers IRW and IR. Each bit position in those binary identifiers represents the other instance of shared resources. The binary identifier IRW_i represents all shared resources used by task t_i that are read (does not change shared resource state) or written (changes shared resource state). The binary identifier IR_i represents all



FIFO$^n_1$ t_{25} t_{19} t_{17} t_9 $\in C^n_1$
FIFO$^n_2$ t_{16} t_1 $\in C^n_2$
FIFO$^n_3$ t_{13} t_6 t_{15} t_{14} t_3 $\in C^n_3$
FIFO$^n_4$ t_{21} t_{20} t_{12} t_4 $\in C^n_4$
FIFO^n_N t_{24} t_{22} t_{18} t_5 $\in C^n_N$
Waiting tasks in class queues

Figure 1: Task classes with queues.

shared resources used by task t_i that are only read.
According to the binary identifiers IRW and IR tasks are grouped into classes. Each requested task has to be assigned to one of task classes, when task class do not exist the new one is created. A single class C_k (where k is class identifier) groups all tasks that have the same values of binary resource identifiers IRW and IR (def. 1).

$$C_k = \{t_i : IRW_i = IRW_k \wedge IR_i = IR_k\} \tag{1}$$

Therefore the task class C_i represents group of shared resources with operation set that is performed on them (with a distinction between read-write, write-only or read-only on each used shared resource). Each class C_i has also individual FIFO queue Q_i, which sets the local execution order between tasks belonging to this class. Sample of task classes queues were shown in figure 1, where each superscript index represents $n - th$ point in time.

$$IRW_i and IRW_j = 0 \tag{2}$$

There is no resource conflict between tasks t_i and t_j if condition 2 is satisfied. Otherwise condition 3 has to be verified to detect resource conflict between tasks. Verification of the condition 2 is recomended only due to performance reason, because it is simplified in comparison with condition 3.

$$(IRW_i and IRW_j) xor (IR_i and IR_j) \neq 0 \tag{3}$$

And at least one resource conflict between tasks t_i and t_j occurs, only if condition 3 is satisfied. This condition defines also outlier as conflicted tasks. Applied outlier detection in events using conditions 2 and 3 are used to create and update the conflict array M^n. This data structure M^n is a representation of Wait For Graph and stores all detected resource conflicts between existed classes in $n - th$ point in time. In the fields of the conflict array instead of storing a nonzero value determined by condition 3, the logical time value of longest waiting task is stored (only for task from fixed row class). This nonzero value represents at least one resource conflict between task classes selected by the intersection of the row and the column in M^n. Otherwise zero value is stored at the intersection of row and column of conflict array, this stored zero value represents no resource conflicts between those tasks classes. The conflict array is required to prepare conflictless schedule S^n, that determines from which class queues the tasks can be resumed to execute them in parallel. Using the conflict array avoids duplications of the same calculations every time, when conflictless schedule S_n was prepared. It is worth to mention, that prepared conflictless schedule S^n is correct only for $n - th$ point in time, because it is corresponding to specific state of task processing environment. Time points are determined by events that report a new task or finish one of active tasks. Therefore right conflictless schedule has to be determined in each point in time, their assembly creates adaptive conflictless task schedule S. The whole problem of conflictless scheduling bases on proper management of class queues, where is no occurrences of resource conflicts between active tasks.

1.2. Adaptive conflictless task schedule

The adaptive conflictless scheduling fixes execution order for tasks involved in shared resource conflict and in this way assures mutual exclusion between tasks in access to shared resources. Presented scheduling takes into account all currently executed tasks in parallel, that define set of active tasks R^n for $n - th$ point in time. The usage of adaptive conflictless scheduling requires the assumption that for each task all set of its shared resources are known in advance and this resource set is constant. The software developer is obligated to select in program code each task boundaries, that each selected sequence of instructions specified as a task was as short as possible [6]. This is similar to separation of critical sections in program, what is often used in concurrent programming. After marking tasks boundaries in program the conflitless scheduling can coordinate their executions whereby some of them, that are involved in shared resource conflict, have to be delayed. To prevent task starvation the fairness rule is applied, so from two tasks that have resource conflict the younger one is suspended. The suspended task is waiting for execution in FIFO queue of class, to which this task belongs. However conflictless scheduling guarantees for any suspended task that its execution starts in finite time [11].

The creation of conflictless schedule for $n - th$ point in time requires to prepare an conflictless schedule for each situation, when one of active task from R^n will be finished. The number of prepared conflictless schedules for $n + 1$ point in time is determined by number of tasks in active task set R^n. This is the result of assumption that execution time of any active task is unknown in advance. Of course, active task finish has to be reported, even this execution was finished with error. The completed task free all used resources, so other waiting conflicted tasks can be executed according to prepared conflictless schedule. Others conflictless schedules prepared for the same moment of time are useless. Additional structures like task classes and conflict array, that were presented in previous chapter, are required to effective prepare conflictless schedule. The outlier detection has to be used to assure consistency between required data structures and current state of task processing environment.

Each prepared S_k^n conflictless schedule determines classes from that tasks can be released from FIFO queues, when active task from class C_k finish as a first its execution in $n - th$ point in time. According to class definition and values of its binary resource identifiers some of them release only the oldest waiting task, but the other class can release all waiting tasks from queues [6]. All tasks that release class queues starts its execution immediately, after that they belongs to active set R^n. The class, that task was released from its queue and starts its execution, will be marked as active. In $n - th$ point in time environment state includes set of active task R^n that are executed and all tasks waiting for execution in C_k^n class FIFO queue where $k = \{1, ..., N^n\}$, N^n represents number of task classes in $n - th$ point in time. Each finish of execution for one of active tasks determines next points in logical time. The next $(n + 1) - th$ point in time can not be determined in advance, because task execution time is unknown before it is finished.

The algorithm for preparation conflictless schedule assures fairness and liveness for all waiting tasks. The prioritization of tasks is not supported in conflictless scheduling. To prevent occurrence of the task starvation problem in conflictless scheduling each task besides of two binary resource identifiers IRW and IR has assigned a timestamp of logical time. The conflictless schedule preparation includes checking of logical time to determine the execution order for waiting conflicted tasks. According to this rule the oldest conflicted task will be executed first. Applied rule prevents from the situation, that in next prepared adaptive conflictless schedules tasks from fixed set of task classes begin execution, but other task classes queues are never emptying. In some situations (some environments state) many alternative conflictless schedule can exist. Then the arbitration rule choose one of these alternative schedules. The arbitration rule prevents task starvation, because it chooses to execution a subset with the oldest conflicted waiting task and that subset of tasks has no resource conflict with each other. This assures that in conflictless schedule preparation the oldest waiting task will be preferred than many other task, that have conflict with the oldest task. This eliminate frequently repeated task set in conflictless schedule and assures that each waiting task will be executed in finite time. Taking into schedule all requested tasks causes many exceptional situations. It is important to note, that adaptive conflictless schedules have to be prepared often and in parallel, because set R^n has usually many elements. Also minimization of delays related with computation group of conflictless schedules is important, therefore isolated computing environment GPGPU (General Purpose computing on Graphic Processing Unit) was used. The most important feature of GPGPU is massively parallel processing using GPU (Graphic Processing Unit) with its memory and computing units. Using GPU as SIMD (Single Instruction Multiple Data) architecture for scheduling task in SISD (Single Instruction Single Data) is not novelty, ealier GPU was used for efficient schedule preparation for transaction processing [12, 13, 14]. How to use GPU to create conflict array and determine adaptive conflictless schedule with example performance results were presented in other publications [5].

2. Resolving classical concurrency problems using adaptive conflictless scheduling with outlier detection

The adaptive conflicteless scheduling in universal approach and can be used to to solve many concurrency problems. The programmer using conflictless scheduling does not have to choose dedicated synchronization mechanisms or algorithm for allocation shared resources to executed task. The conflictless scheduling guarantees that task execution is realized without interruption due to resource access. For every one considered in subchapters classical concurrency problem that was solved using conflictless scheduling will be presented:

- task boundaries,
- binary resource identifiers assigned for tasks.
- task classes and conflict array,

In all considered scenarios the binary resource identifiers are statically assigned for tasks, also created conflict array is static. All presented concurrency problem are solved by using adaptive conflictless scheduling in standard and extended configuration of environments.

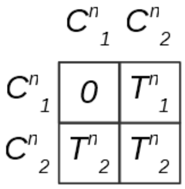
2.1. Readers and writers concurrency problem

The classical readers and writers concurrency problem was formulated by Courtois et al. [15]. This classical problem will be presented in two variants: standard and extended, in both of them the adaptive conflictless scheduling will be used to solve them.

2.1.1. Solution of standard concurrency problem of readers and writers

In this concurrency problem definition the processing environment has two type of tasks: readers and writers and single instance of shared resource, known as a data source. Tasks belonging to readers group are defined as read operation on the data source. Whereas tasks from writers group determines write operation, which change state of the data source. It have to be noted, that parallel executions of writer task with any other task is not possible due to resource conflict, but parallel executions of reader tasks is permitted.

According to task definition in classical readers and writers concurrency problem, there will be distinguished two task classes: readers task class C_r and writers task class C_w. For determined two task classes resource conflict are detected using presented outlier detection and stored in conflict array. The conflict array created for those classes is static, so no updates of this data structure occurs in further processing. Each new requested reader or writer task has to be verified, to which class this task belong. Then basing on conflict array shown on the figure 2 the set of active tasks R^n is determined for any $n-th$ point in time. Those active tasks can be executed in parallel without resource conflict. Therefore for $n-th$ point in time only one from two classes can be active and the active class has always at least one executed task. Always for $n-th$ point in time only single conflictless schedule has



	C^n_1	C^n_2
C^n_1	0	T^n_1
C^n_2	T^n_2	T^n_2

Figure 2: Conflict array for standard readers and writers concurrency problem.

to be prepared, even in situation where there are many active tasks. This is due to the fact, that in readers and writers problem all active tasks belong always to the same class.

The control which of two FIFO queues, that are related respectively with task class C_w and C_r, should be emptying base on the prepared conflictless schedule. For readers task class C_r binary resource identifiers $IRW_r = IR_r$, so executions of all waiting in this queue Q_r readers tasks will be resumed at once. For writers task class C_w, where resource binary identifiers $IRW_w \neq IR_w$, therefore only the longest waiting task can leave class queue and start execution. The longest waiting task can be determined in simple way, because each queue for task class is managed by FIFO algorithm.

All resources conflict reveals conflict array, which is presented of figure 2. Prepared conflictless schedule for $(n+1)-th$ point in time bases on detected resource conflicts between classes and analysis of logical time values determined for the oldest waiting tasks in each class queues that has resource conflict with finished active task. Preparation of conflictless schedule for $(n+1)-th$ point in time have to be done for two mutually exclusive cases. The first case, when active task from set R^n have one or many readers tasks. The second case, where the active task set R^n has only single writer task.

Independently for both cases, if the oldest waiting task belongs to writers class C_w, then after finished execution of active tasks the execution of writer task will be started. Otherwise, if the oldest waiting task belongs to readers class C_r, then more readers tasks will be executed in parallel. This choice is determined by time values of oldest waiting task in queues Q_r and Q_w denoted respectively as T^n_r and T^n_w. If $T^n_w > T^n_r$ then after finished active writer task another writer task begin execution, otherwise all waiting readers tasks start execution after writer task execution will be finished. Those choice prevents task starvation, which theoretically could occur in situation, when only tasks from the same class will be run. Adaptive conflictless

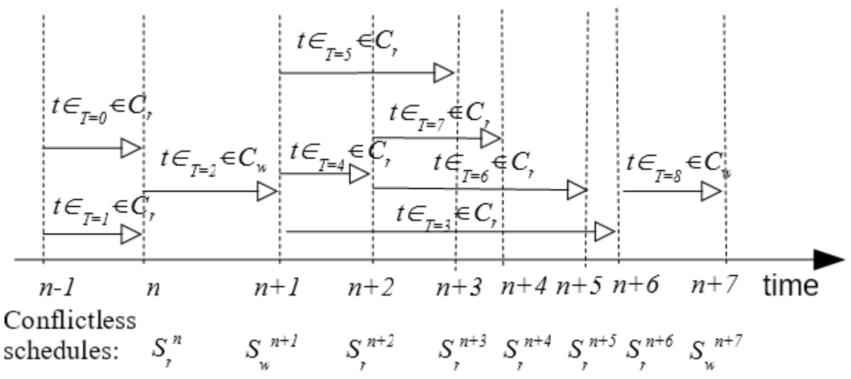


Figure 3: The example of adaptive conflictless schedule for standard readers and writers tasks.

schedule guarantees that tasks from all class queues will be regularly released. This ensures liveness of any task that is controlled by adaptive conclitless schedule.

It should be noted that all finished task executions should be notified, even where this is the result of the task processing error. Adaptive conflictless schedule is constructed as a sequence of selected conflictless schedules, each of them is chosen according to finished execution one of active tasks. Every finish of active task fixes time points, in which according to the prepared conflictless schedule tasks executions are started.

The figure 3 presents sample adaptive conflictless schedule determined for readers and writers concurrency problem. Presented tasks belongs to readers class C_r or writers class C_w and task subscript index shows its logical time value. It should be noted, that some of presented in figure 3 conflictless schedules are empty collections i.e. S_r^{n+3}, S_r^{n+4}, S_r^{n+5}. This is a result of logical time verification for waiting tasks. It is also shown, that many tasks from readers class C_r can be executed in parallel, but when writer task is executed, all other tasks have to wait.

The conflictless scheduling guarantees that there is no resource conflict between parallel executed tasks. Each active task can execute without any additional delays associated with obtaining access to the required resources. The suspension of conflicted task causes additional delay, but this is the result of elimination of resource conflicts between readers and writers.

$$\begin{array}{c|c|c|c|c|c|c|}
 & C^n_{r1} & C^n_{r2} & C^n_{r3} & C^n_{w1} & C^n_{w2} & C^n_{w3} \\
\hline
C^n_{r1} & 0 & 0 & 0 & T^n_{r1} & 0 & 0 \\
\hline
C^n_{r2} & 0 & 0 & 0 & 0 & T^n_{r2} & 0 \\
\hline
C^n_{r3} & 0 & 0 & 0 & 0 & 0 & T^n_{r3} \\
\hline
C^n_{w1} & T^n_{w3} & 0 & 0 & T^n_{w1} & 0 & 0 \\
\hline
C^n_{w2} & 0 & T^n_{w3} & 0 & 0 & T^n_{w2} & 0 \\
\hline
C^n_{w3} & 0 & 0 & T^n_{w3} & T^n_{5} & 0 & T^n_{w3} \\
\hline
\end{array}$$

Figure 4: Conflict array for extended readers and writers concurrency problem with three data sources.

2.1.2. Solution of extended concurrency problem of readers and writers

In extended concurrency problem definition the types of tasks remain unchanged, but there are many independent data sources in task processing environment. Each task has statically assigned data source on which performs its read or write operation. The number of data sources limits the length of binary resource identifiers *IRW* and *IR*.

Then in extended problem definition, the number of task classes doubles the number of data sources that exists in extended environment of readers and writers task processing. Reader tasks classes are marked as C_{r1}, C_{r2}, ... C_{rN} and for writers tasks classes are indicated as C_{w1}, C_{w2}, ... C_{wN}, where N is number of data sources. The number in class subscript represents data source identifier, so resource conflict occurs between each classes C_r and C_w with the same subscript number. All conflict dependencies between classes are presented in conflict array prepared for extended classical concurrency problem definition of readers and writers (fig. 4).

Just like in standard classical concurrency problem, definition management of class queues bases on adaptive conflictless scheduling. For $n-th$ point in time conflictless schedules are prepared, each one for situation when other one active task finish execution. However in extended problem definition active tasks can come from different classes. It is even possible that active tasks from set R^n include

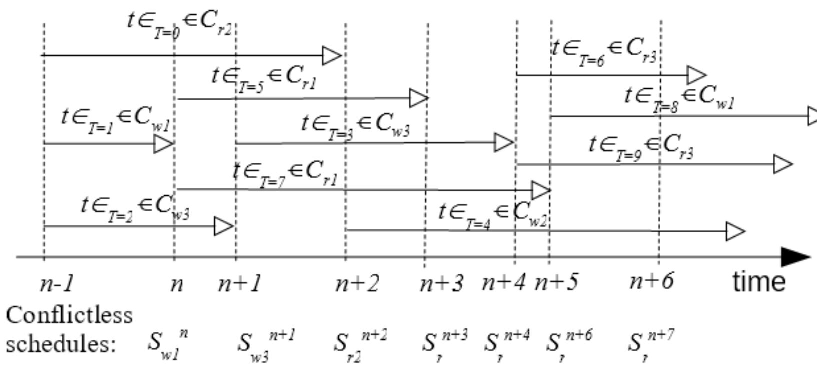


Figure 5: The example of adaptive conflictless schedule for extended readers and writers tasks with three data sources.

both readers and writers tasks, if only they perform operations on different data sources. Example of adaptive conflictless schedule for three data sources presents figure 5.

2.2. Dining philosophers concurrency problem

The classical dining philosophers problem was formulated by Dijkstra, originally as five computers competing for five tape devices. Dining philosophers problem was reformulated by Hoare at al. in present form [15, 16].

2.2.1. Solution of standard concurrency problem of dining philosophers

The classical dining philosopher concurrency problem is defined for five philosophers which sit at a round table with forks. Each fork lies between two other neighboring philosophers. Every philosopher has to change its state form eating to resting and then again from resting to eating. The philosopher state changes represents its processing progress, in any state philosopher stays only in finite time. Before eating each philosopher requires to access two nearest forks, each of them is exclusive resource. After eating philosopher immediately releases two forks (they are returned on table) and then philosopher changes his state to resting. In resting state philosopher does not require any resources.

To use conflictless scheduling task has to be defined as philosopher eating phase and forks are examples of shared resources. The philosopher represents a

$$
\begin{array}{c|c|c|c|c|c|}
 & C^n_1 & C^n_2 & C^n_3 & C^n_4 & C^n_5 \\
\hline
C^n_1 & T^n_1 & T^n_1 & 0 & 0 & T^n_1 \\
\hline
C^n_2 & T^n_2 & T^n_2 & T^n_2 & 0 & 0 \\
\hline
C^n_3 & 0 & T^n_3 & T^n_3 & T^n_3 & 0 \\
\hline
C^n_4 & 0 & 0 & T^n_4 & T^n_4 & T^n_4 \\
\hline
C^n_5 & T^n_5 & 0 & 0 & T^n_5 & T^n_5 \\
\hline
\end{array}
$$

Figure 6: The conflict array for standard concurrency problem of five dining philosophers

sequence of tasks that requires identical set of resources. Next task reported by the same philosopher requires finish of resting phase, which in turn requires finish of previously reported by him task. According to concurrency problem definition each fork is shared only by tasks reported by two neighboring philosophers. Therefore never two tasks reported by neighbor philosophers can eat simultaneously.

In concurrency problem of dining philosophers the number of resource groups is identical with the number of philosophers. Therefore in solution of five dining philosophers using conflictless scheduling only five resource class presents. The minimal length of binary resources identifiers is determined by the number of used shared resources, which is always identical with the number of philosophers. Each task class C_k represents group of two forks, which are identified in IRW_k binary representation. Example of binary resource identifiers values for five task classes $C_k, k = 1..5 : IRW1 = (10001)_b, IRW2 = (11000)_b, IRW3 = (01100)_b, IRW4 = (00110)_b, IRW5 = (00011)_b$. The second binary resources identifier $IR_k = 0$ for $C_k, k = 1..5$, because in this concurrency problem there is no shared resources that are accessed in read only manner. Therefore resource conflict detection between two task classes is verified using the simplified condition 2.

The conflict array for standard dining philosophers problem is determined for task classes $C_k, k = 1..5$ and this structure will be used in preparation of conflictless schedules. The determined conflict array in $n - th$ point in time for standard problem of five dining philosophers is shown on the figure 6. Each class queue has maximally one task. In $n - th$ point in time the state of task class FIFO queue is denoted Q_k^n and the longest waiting task in FIFO queue Q_k^n is marked as T_k^n.

The active task set R^n can include tasks belonging to many classes, except for the task couples that are from directly next classes. To prevent any task starvation in FIFO queue for each active class C_k^n the class C_k^{*n} has to be determined that FIFO queue includes the oldest conflicted task. Determination of the class C_k^{*n} requires to find the lowest value of logic time T_k^n in column of conflict array that is assigned to the active class C_k^n. When class C_k^{*n} is determined for each active class $C_k^n, k = 1..count(R^n)$, then any conflictless schedule S_k^{n+1} does not include any task, that class has resource conflict with any one of determined classes C_k^{*n}. This rule assures liveness for all tasks requested by philosophers. Therefore the deadlock situation, when five parallel tasks reserve only one fork, never occurs.

The example adaptive conflictless schedule for standard concurrency problem of dining philosophers problem presents the figure 7. On this figure each task is representing eating phase of selected philosopher and there is no representation of resting phase for dining philosophers. The presented subscript index for task determines its logical time value and its class C_k subscript represents philosopher identifier $k = 1..5$. Some of conflictless schedules presented on the figure 7 is empty i.e. S_3^{n+3}, S_2^{n+6}. The schedule S_1^{n+4} includes only task from class C_2^{n+4} because waiting task execution from class C_4^{n+4} is blocked by running conflicted task from active class C_5^{n+4}. Therefore execution of task from class C_4^{n+4} is running just in $n + 5$ point in time after finished execution of task from class C_5^{n+4}.

Another important aspect in conflictless scheduling presented on the figure 7 is the number of conflictless schedules that have to be prepared for each point in time. For $n + 1$ point in time two schedules S_2^{n+1} and S_4^{n+1} should be prepared, but only one of them will be chosen to execution. For $n + 2$ point in time also two conflictless schedules need to be prepared S_2^{n+2}, S_5^{n+1} and for $n + 3$ point in time three conflictless schedules are required S_1^{n+3}, S_3^{n+3}, S_5^{n+3} e.t.c.. Generally from prepared conflictless schedules for $n - th$ point in time only one is used. It should be noted, that running task sequence is not always consistent with order of its logical time values. In $n + 1$ point in time execution of task from class C_5^{n+1} is started, that has logical time value lower than task from class S_3^{n+2} which execution is started later. According to conflictless scheduling in some environment situations longer waiting task can be executed later that task from other class.

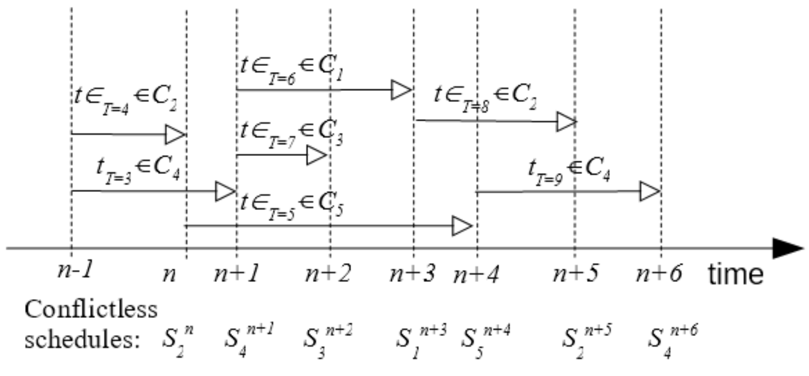


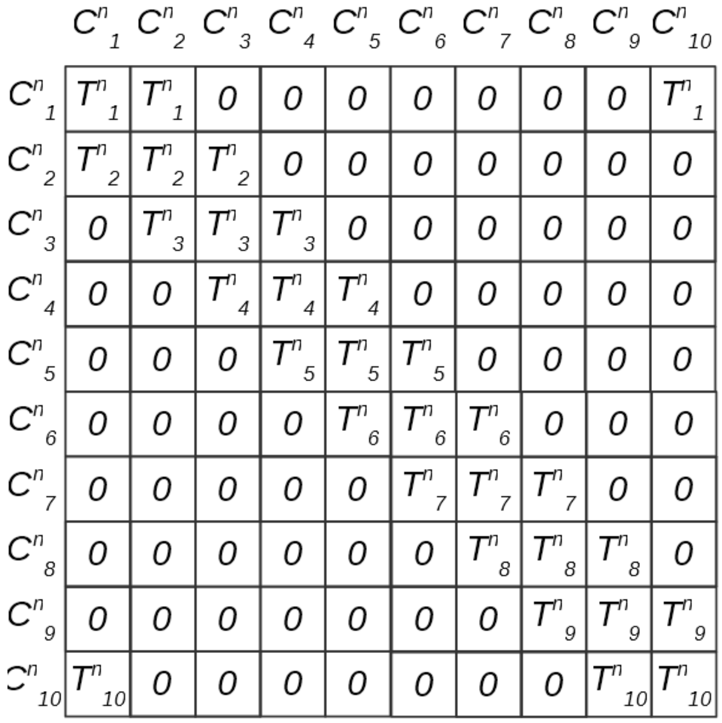
Figure 7: The example of adaptive conflictless schedule for standard concurrency problem of five dining philosophers.

2.2.2. Solution of extended concurrency problem of dining philosophers

Extended concurrency problem concerns the even number of philosophers. Minimal environment settings considers two philosophers with two shared forks located on table. Tasks are defined identically as presented in standard definition of this concurrency problem, but task reported by other philosophers compete for those two forks. Those forks have to be accessed in mutually exclusive manner. Interesting is the fact that in this minimal problem definition environment there is only single task class. This single class FIFO queue determines the tasks executions order, which is identical with task reporting order. Therefore this minimal environment settings of dining two philosophers is trivial to analysis adaptive conflictless scheduling.

In extended variant of dining philosophers problem will be considered case with ten philosophers. In this environment in opposite to the minimal problem definition many tasks can be active. Each of philosophers reports tasks that represents its eating phase. Tasks reported by the differ philosophers has other shared resource group, therefore each philosopher has own task class. In this task processing environment usage of adaptive conflictless scheduling assures correctness of task processing, lack of task starvation and no deadlock.

The conflict array presented on figure 8 has been prepared for ten task classes. Any FIFO queue from that task classes holds tasks reported by the same philosopher. Assuming regular task reporting and fixed resting and eating time for all



	C^n_1	C^n_2	C^n_3	C^n_4	C^n_5	C^n_6	C^n_7	C^n_8	C^n_9	C^n_{10}
C^n_1	T^n_1	T^n_1	0	0	0	0	0	0	0	T^n_1
C^n_2	T^n_2	T^n_2	T^n_2	0	0	0	0	0	0	0
C^n_3	0	T^n_3	T^n_3	T^n_3	0	0	0	0	0	0
C^n_4	0	0	T^n_4	T^n_4	T^n_4	0	0	0	0	0
C^n_5	0	0	0	T^n_5	T^n_5	T^n_5	0	0	0	0
C^n_6	0	0	0	0	T^n_6	T^n_6	T^n_6	0	0	0
C^n_7	0	0	0	0	0	T^n_7	T^n_7	T^n_7	0	0
C^n_8	0	0	0	0	0	0	T^n_8	T^n_8	T^n_8	0
C^n_9	0	0	0	0	0	0	0	T^n_9	T^n_9	T^n_9
C^n_{10}	T^n_{10}	0	0	0	0	0	0	0	T^n_{10}	T^n_{10}

Figure 8: The conflict array for extended concurrency problem of dining philosophers

philosophers, the FIFO queues from task classes will be emptied alternately. This can be observed in example adaptive conflictless schedule presented in figure 9.

If philosophers eating or resting time is different, then in the particular case adaptive conflictless scheduling two tasks from the same class can be executed sequentially. This means that one of philosophers has two eating phases separated by resting phase, when other was in resting phase. Also in this environment there will be no task starvation. This fairness rule is a result of taking into account in the conflictless schedule preparation a unique values of logical time, that are assigned individually to tasks reported by philosophers.

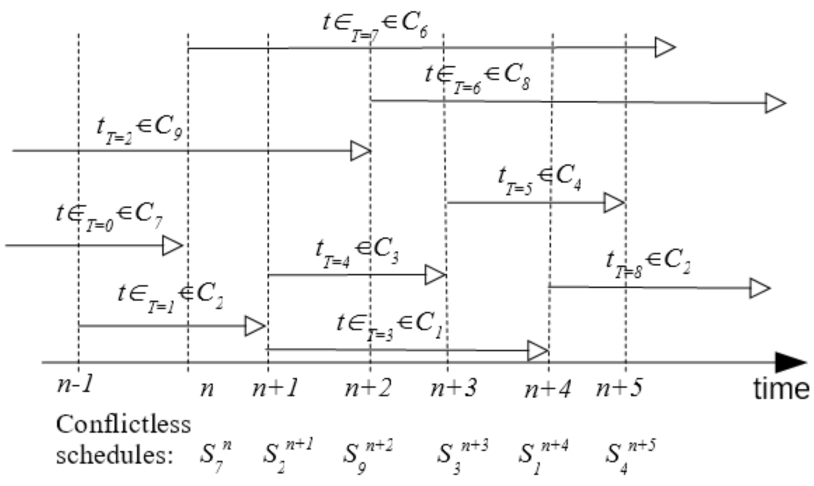


Figure 9: The example of adaptive conflictless schedule for extended concurrency problem of dining philosophers.

3. Conclusions

Event space analysis in task processing environments allows detection of possible outliers, each one is identified as resource conflict. The outlier detection bases on dedicated model of task resource representation, this allows to rapid detection resource conflict. The outlier detection was applied to prepare conflict array for task classes, that is required to effective preparation of task schedule without resource conflict occurrences between tasks executed in parallel. Presented outlier detection also help to eliminate deadlocks.

The conflictless schedule provides alternative coordination method of tasks processed in high contention environment with limited number of shared resources. In presented approach task definition is flexible, therefore software developer can separate tasks in various task processing environments and prepare adaptive conflictless schedule. The adaptive conflictless scheduling was applied to eliminate unnecessary events like resource conflicts to improve effectiveness of parallel tasks processing, as a number of completed tasks in a unit of time.

This novel scheduling approach bases on management of FIFO queues, where each one belongs to other task class. Task class represents group of shared resources and way how task access them with distinction of read/write operations.

Tasks waiting for execution in the same queue have identical shared resource access pattern. This means, that execution each of them requires the same group of shared resources with identical operations of those resources. For task that leaves FIFO queue its execution is started immediately and becomes an active task. Therefore adaptive conflictless scheduling bases on resource group allocation to tasks. The adaptive conflictless schedule is a series of conflictless schedules, where each one of them was prepared for next point in time. Those time points are determined by next completed, active tasks. The number of simultaneous calculated schedules for selected point in time is dependent of the number of active tasks.

The advantage of usage adaptive conflictless scheduling is simplification of software developer work, because programmer only need to select in program code tasks boundaries and set binary identifiers representing used by task shared resources. If adaptive conflictless scheduling is used, there is no need to use any other synchronization mechanisms for control tasks access to groups of shared resources. Another advantage of adaptive conflictless scheduling is the versatility of applications, because it can be used to solve various concurrency problems. This approach was presented to solve two classical concurrency problems: readers and writers and dining philosophers. Each of the two classical concurrency problem was presented in two variants: the standard which has environment configured as in problem definition and the extended environment with specific environment configuration. The extended environment configurations have complex dependencies between tasks and shared resources as standard problem definition. Extended variants of presented concurrency problems especially exposes the task starvation in class queue and deadlock. Application of adaptive conflictless scheduling in both classical concurrency problems provides in each of its variants correct order of task execution without resource conflict. Therefore presented approach is universal and flexible because its adapts to different variants of various task processing environments. In readers and writers problem the number of data sources was increased. In dining philosophers problem usage of adaptive conflictless schedule is possible regardless from the number of philosophers. Analysis of scheduling rules, when conflictless scheduling is applied, shows lack of task starvation and avoidance of deadlock in both presented classical concurrency problems. The adaptive conflictless scheduling also assures liveness and fairness for any controlled tasks. Each of the class queues is regularly release tasks, it is performed in the fair manner. In particular, this is evident in task processing environments with small number of classes, like presented standard variant of readers and writers problem.

References

[1] Tanenbaum, A. S. and Bos, H., *Modern operating systems*, Prentice Hall Press, 2014.

[2] Stallings, W., *Operating systems, Internals and Design Principle*, Pearson Education, 2015.

[3] Pun, K. and Belford, G. G., *Perfornance Study of Two Phase Locking in Single-Site Database Systems*, IEEE transactions on software engineering, , No. 12, 1987, pp. 1311–1328.

[4] Bernstein, P. A. and Newcomer, E., *Principles of transaction processing*, Morgan Kaufmann, 2009.

[5] Smolinski, M., *Coordination of parallel tasks in access to resource groups by adaptive conflictless scheduling*, In: Beyond Databases, Architectures and Structures. Advanced Technologies for Data Mining and Knowledge Discovery, Springer, 2015, pp. 272–282.

[6] Smoliński, M., *Conflictless task scheduling concept*, In: Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology–ISAT 2015–Part I, Springer, 2016, pp. 205–214.

[7] Duraj, A., *Conflictless Task Scheduling Using Association Rules*, In: Beyond Databases, Architectures and Structures. Advanced Technologies for Data Mining and Knowledge Discovery, Springer, 2015, pp. 283–292.

[8] Duraj, A. and Szczepaniak, P., *Information Outliers and Their Detection*, Information Studies and the Quest for Transdisciplinarity: Unity through Diversity, Vol. 9, 2017, pp. 413.

[9] Chomatek, L. and Duraj, A., *Multiobjective genetic algorithm for outliers detection*, In: INnovations in Intelligent SysTems and Applications (INISTA), 2017 IEEE International Conference on, IEEE, 2017, pp. 379–384.

[10] Emets, V. and Rogowski, J., *Scattering of acoustical waves by a hard strip and outlier phenomenon*, In: INnovations in Intelligent SysTems and Applications (INISTA), 2017 IEEE International Conference on, IEEE, 2017, pp. 376–378.

[11] Smoliński, M., *Elimination of task starvation in conflictless scheduling concept*, Information Systems in Management, Vol. 5, No. 2, 2016, pp. 237–247.

[12] Bakkum, P. and Skadron, K., *Accelerating SQL database operations on a GPU with CUDA*, In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM, 2010, pp. 94–103.

[13] He, B. and Yu, J. X., *High-throughput transaction executions on graphics processors*, Proceedings of the VLDB Endowment, Vol. 4, No. 5, 2011, pp. 314–325.

[14] Smoliński, M., *The GPU performance in coordination of parallel tasks in access to resource groups without conflicts*, Information Systems in Management, Vol. 6, 2017.

[15] Courtois, P.-J., Heymans, F., and Parnas, D. L., *Concurrent control with "readers" and "writers"*, Communications of the ACM, Vol. 14, No. 10, 1971, pp. 667–668.

[16] Dijkstra, E. W., *Hierarchical ordering of sequential processes*, In: The origin of concurrent programming, Springer, 1971, pp. 198–227.