Path Expressions Revisited

Thomas Alexander Hövelmann
thomas-alexander.hoevelmann@tu-dortmund.de
Department of Computer Science, Technische Universität Dortmund

Alexander Krause
alexander.krause@tu-dortmund.de
Department of Computer Science, Technische Universität Dortmund

Peter Ulbrich
peter.ulbrich@tu-dortmund.de
Department of Computer Science, Technische Universität Dortmund

Abstract

Path expressions provide a precise and concise method for defining synchronization rules for accessing shared data. In conjunction with predicates, even finer control of the sequences can be achieved. Furthermore, path expressions, with their ability to abstract from a specific synchronization technique, demonstrate a high level of adaptability. Despite being developed nearly five decades ago, they remain underrepresented in current software engineering and have yet to achieve widespread applicability. This paper reevaluates existing approaches and presents an automaton-based version of path expressions written in C++. We extend the concept of predicates to allow access to external system states and evaluate the feasibility and performance of our approach.

Keywords

Synchronization, Path Expressions, Parallelism, Predicates

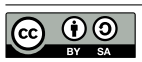
1 Introduction

The advent of multi- and many-core CPUs has created many synchronization primitives, as evidenced by the literature (e.g.,[10, 11, 13]). For instance, the pthreads library offers five primary synchronization techniques (i.e., mutexes, condition variables, read-write locks, spinlocks, and barriers), while popular Python extends the options available to developers to six primary primitives[1] plus queues and multiprocessing variants. This is in addition to the possibilities offered by contemporary OS kernels, such as Linux [10] or FreeBSD [11], or even non-blocking synchronization techniques [12]. Basic synchronization primitives, such as semaphores, are often the building blocks for constructing complex synchronization patterns, including those that tackle the reader-writer problem [13]. This process, while common, is not without its challenges, underscoring the need for advanced techniques. This process is time-consuming and susceptible to errors, resulting in the generation of considerable code for each pattern. The programmer's primary concern is the correctness and efficiency of the synchronization implementation. They aim to express both the steps involved in accessing a shared memory and the degree of parallelism.

One potential solution to this issue is path expression-based synchronization, which is the focus of this paper. Initially introduced as early as 1974 by Campbell and Habermann [3], the concept represents an abstract description language for the specification of execution sequences. Based on these classic path expressions, some extensions were proposed to increase the expressive power of the path expressions. These extensions were put forth by Andler (1979), Habermann (1975), and Flon Habermann (1976) [1, 5, 7], respectively. Of particular note is Andler's work on predicate-based path expressions, which endeavors to consolidate all situational extensions into a single concept: introducing predicates. In this context, the term "predicate" refers to a concept derived from predicate logic, in which predicates represent the truth-functional component of atomic statements. The evolution of path expressions continued with the introduction of the path-process notation by Lauer and Campbell (1975) [8], which later merged into the COSY notation developed by Lauer, Torrigiani, and Shields (1979) [9]. However, both approaches lack an operator for explicit parallelism, as provided by the preceding approaches. It is also worth noting that path expressions seem to be rarely used in practice, although there have been previous implementations in Pascal [2], proposals to introduce them in C++ [14], and an orphaned Python GIT project [6] on the subject.

In light of the potential advantages of path expression-based synchronization, this paper seeks to address the reasons for this discrepancy and to elucidate the potential benefits of path expression-based synchronization. The contributions of this paper are as follows:

[1] https://docs.python.org/3/library/asyncio-sync.html



 DOI: https://dx.doi.org/10.1145.nnnnn. GI/ITG FG Operating Systems, October 2024, Stuttgart, DE.

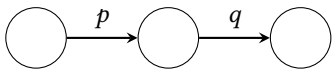
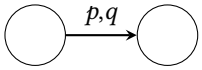
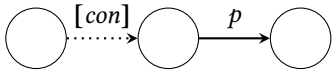
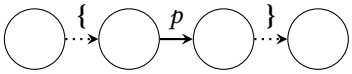
symbol	automaton
$p\,;\,q$	
$p\,,\,q$	
p^*	
$p[con]$	
$\{p\}$	

Table 1: Syntax of automaton-based path expressions: p and q indicate procedures, *con* predicate, **;** sequence operator, **,** selection operator, * repetition operator, **[]** restriction operator, **{}** braces operator.

- Introduction of automata-based path expressions derived from predicate-based path expressions,
- extension of the existing concept of predicates and
- evaluation of the approach above regarding expressiveness, usability, and performance.

2 Approach

We adopted the predicate path expressions methodology, initially proposed by Andler [1], for this work. We refer to the resulting approach in the following as *automaton-based path expressions*. This approach, which we've tailored to our needs, is adaptable and can be applied to a wide range of scenarios. We employ five of the six operators described in Andler's approach, namely, the selection operator, the sequence operator, the repetition operator, the restriction operator, and the braces operator excluding the collateral operator, as its function can be achieved through the use of the other operators. Our semantics of the operators are analogous to that of Andler's approach, with the exception of the restriction operator. We've enhanced this operator by incorporating an expansion regarding the potential predicates, which we believe significantly improves its functionality. In addition to the *req*, *act*, and *term* counters and constants in predicate path expressions as described by Andler, external integer-compatible variables may now also be used as references in predicates. This modification enables the dependency of permitted execution sequences on external system states contingent on a specified path expression. To illustrate, the execution of a procedure may be contingent upon the status of a variable that indicates the readiness of a peripheral device utilized during that procedure.

In addition to their symbol representations, we assigned the individual operators an automaton representation as illustrated in Table 1. In the automaton representation, a state represents the current state of the synchronization system. Transitions may be classified into two categories: normal procedure transitions and functional transitions. Functional transitions provide additional functionality to procedure transitions but can never be triggered independently. There are two distinct types of transitions: predicate transitions, which can only be completed if the predicate protecting them is satisfied at the time of the transition, and parallel environment transitions, which indicate the commencement or conclusion of parallel environments. Each valid transition process triggers precisely one procedure transition and unlimited functional transitions. However, each transition process can utilize a maximum of one parallel environment transition, either entering or exiting.

3 Implementation

To analyze the performance of automaton-based path expressions, we developed a C++ prototype. It is comprised of three principal functional units implemented by a dedicated class: `Predicate`, `Automaton`, and `Synchronizer`.

The `Predicate` class represents a recursive data structure that implements the functionality of the predicates. A single object of the class can either process an operand with an unary logical or arithmetic operator or link two operands with a binary logical or arithmetic operator. The possible operators are other Predicate objects or references to variables of an integral type compatible with the C++ standard [4].

The `Automaton` class represents a data structure for storing a state graph with all its transitions. This is specified directly by an automaton-based path expression. The logic is oriented in a manner that is highly consistent with the automaton analogy. Additionally, the automaton maintains a record of the currently active states and provides functions for determining whether a specific procedure is permitted to execute within the current system state. Moreover, it oversees the management of counters, specifically *req*, *act*, and *term*, for each procedure within the path expression. The transitions within an `Automaton` object are predicated upon the use of `Predicate` objects.

The `Synchronizer` class serves as the control structure that implements the actual synchronization. It utilizes an automaton object as a representation of the path expression. A thread may use a function call via the `Synchronizer` object to ascertain whether executing is currently permitted. To this end, incoming threads are placed in a queue and are awaiting processing in the order of their requests. Upon

determination that a woken thread is permitted to execute, the invocation of the `Synchronizer` object is terminated, thereby enabling the critical section to be entered. If the requisite waiting period has not been reached, the thread re-enters the queue at the rear. This unsuccessful checking is regarded as active waiting. A thread that has already been permitted to run then signals the conclusion of its critical section to the `Synchronizer` object by calling another function, which updates the corresponding data structures and activates further threads if necessary.

4 Experiment Setup

We conducted runtime experiments to assess the effectiveness of our prototype, comparing it to the established synchronization methodology. To ensure the relevance of our findings, we designed a complex synchronization scenario that mirrors real-world applications, thereby enhancing the practicality of our evaluation. The example comprises two distinct types of producers or writers (`write1` and `write2`), each capable of accessing a type-specific ring buffer with limited capacity. If there is still free capacity, these writers can store data in the buffer element by element. Additionally, a consumer or reader (`read`) type is present, which reads a minimum number of elements once they have been stored in both ring buffers and releases the capacities accordingly. In the experiment, there are multiple instances of the writers and a single instance of the reader. Consequently, each access to the ring buffer is exclusive, either to an instance of the respective writer type or to the instance of the reader. Such a scenario is conceivable, for example, in a server-client setup where multiple clients provide disparate sensor data, such as temperature and pressure, temporarily stored on the server side and processed into gas volumes by value pairs through a regular server-side process. The ring buffer capacities ($n_{\text{write1,max}}$ bzw. $n_{\text{write2,max}}$) were set to 8 and the minimum number of elements ($n_{\text{read,min}}$) before reading was set to 4. Furthermore, the variables n_{write1} and n_{write2} describe the current capacity in the respective ring buffer. The resulting automaton representation of this synchronization scenario is shown in Figure 1 and given by:

$$\begin{aligned} \text{cond1} \coloneqq\ & act(\text{write1}) = term(\text{write1}) \wedge act(\text{write2}) = term(\text{write2}) \\ & \wedge n_{\text{write1}} \geq n_{\text{read,min}} \wedge n_{\text{write2}} \geq n_{\text{read,min}} \end{aligned} \tag{1}$$

$$\begin{aligned} \text{cond2} \coloneqq\ & act(\text{write1}) = term(\text{write1}) \wedge n_{\text{write1}} < n_{\text{write1,max}} \\ & \wedge \left(req(\text{read}) = act(\text{read}) \vee n_{\text{write1}} < n_{\text{read,min}}\right) \end{aligned} \tag{2}$$

$$\begin{aligned} \text{cond3} \coloneqq\ & act(\text{write2}) = term(\text{write2}) \wedge n_{\text{write2}} < n_{\text{write2,max}} \\ & \wedge \left(req(\text{read}) = act(\text{read}) \vee n_{\text{write2}} < n_{\text{read,min}}\right) \end{aligned} \tag{3}$$

The two primary variables of interest in the experiments were the mean start times of all threads following their creation and the mean end times, or the times at which the final thread terminated. In the course of the experiments, five influencing variables were varied:

(1) The synchronization method (Sync.m.)
(2) The number of threads per type of writer (TpT)
(3) The number of repetitions per thread (RpT)

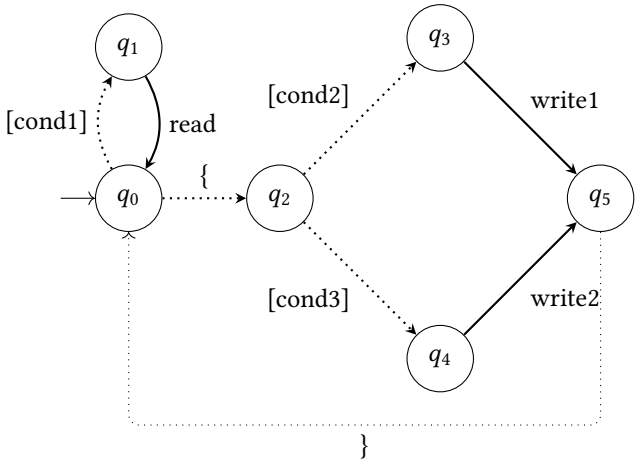


Figure 1: Automaton representation of the synchronization scenario.

(4) The ratio of non-critical iterations (NCIt) to critical iterations (CIt)
(5) The execution mode (Exm.)

The synchronization method could be utilized in two distinct ways: the classic synchronization (C) method, which employs primitives, or the path expression-based (PE) approach, which uses the implemented prototype. The number of TpT represents the number of instances of the two writer types that were created and could be either 1, 10, or 100. The number of RpT describes the frequency with which an instance should repeat its task until its thread is completed. This value was either 1, 10, or 100. The ratio of non-critical iterations (NCIt) to critical iterations (CIt) simulates a workload resulting from active idling outside and within the critical section. The ratios 3,000,000 NCIt : 1,000,000 CIt, 2,000,000 NCIt : 2,000,000 CIt, and 1,000,000 NCIt : 3,000,000 CIt were employed, with the total load remaining constant. The Exm. designation indicates whether the experiment was conducted in a pseudo-parallel (pp) mode on a single CPU core or a parallel (p) mode on three CPU cores.

We conducted 21 experimental repetitions for each of the 108 resulting combinations of influence variables. The total running time was calculated in each instance as the difference between the end and start times. The median of the 21 results was employed as the position parameter for a direct comparison of the running times for C and PE synchronization. The scatter was quantified using the lower and upper quartiles. Furthermore, we derived three ratios to visualize the relationship between the runtimes of the two synchronization methods. The ratios observed from the perspective of PE synchronization are classified according to their relative performance as follows:

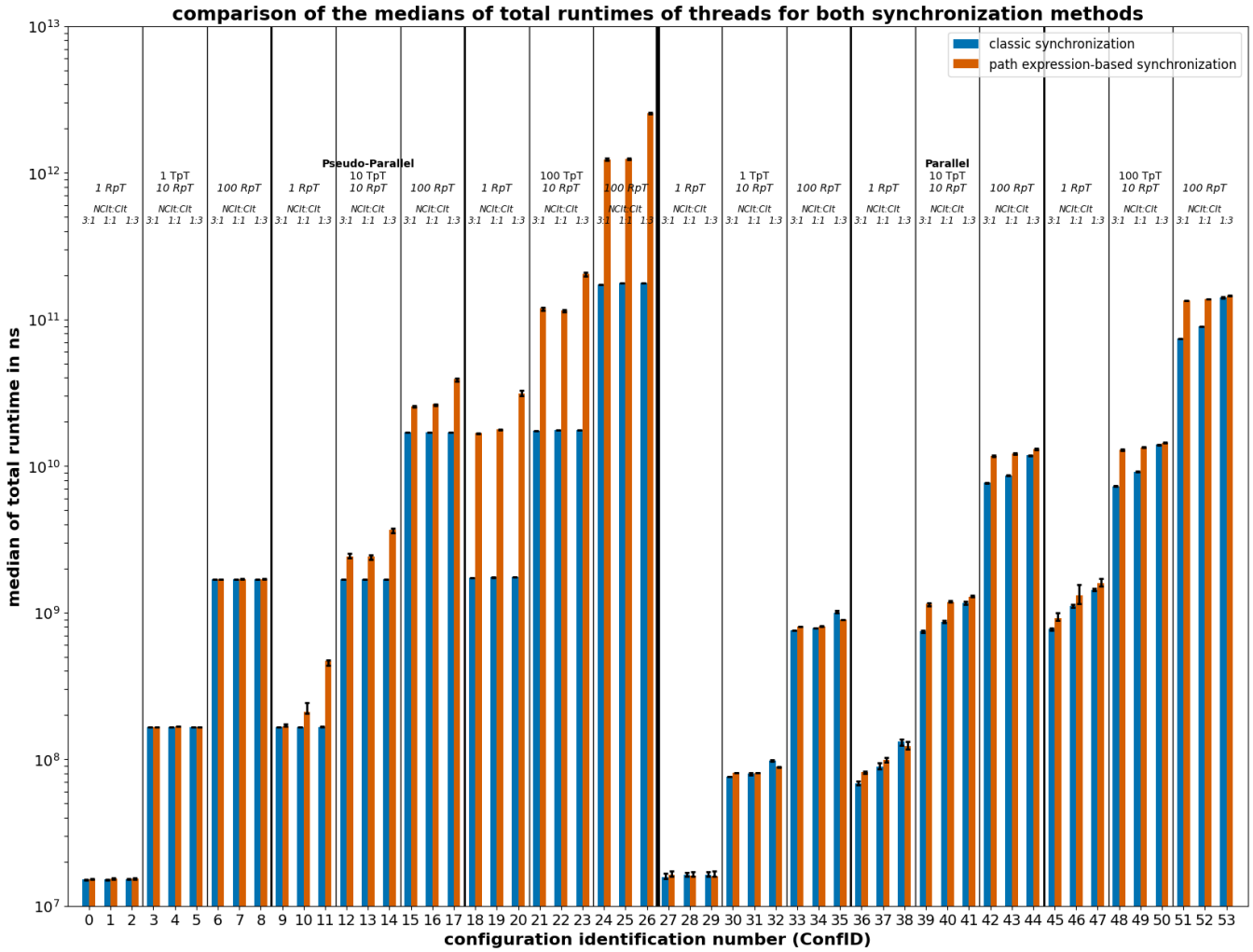


Figure 2: Comparison of runtimes for all threads for both synchronization methods. The values depend on the configuration (*ConfID*), which includes the execution mode {pseudo-parallel, parallel}, threads per type (TpT) {1, 10, 100}, and repetitions per thread (RpT) {1, 10, 100}. Ratio of non-critical to critical iterations (NCIt: CIt): 3:1, 1:1 and 1:3. Median runtimes are plotted with lower and upper quartiles.

- The worst ratio is obtained by dividing the maximum runtime for PE synchronization by the minimum runtime for C synchronization.
- The average ratio is calculated by dividing the median runtime for PE synchronization by the median runtime for C synchronization.
- The best ratio is obtained by dividing the minimum runtime for PE synchronization by the maximum runtime for C synchronization.

5 Results

Figure 2 plots the median running times for C and PE synchronization in nanoseconds on the y-axis. Please note that the scale is logarithmic. The lower or upper error corresponds to the lower or upper quartile. The various experiment configurations are represented on the abscissa. The runtimes for classic synchronization are illustrated as deep blue bars, while those for path expression-based synchronization are represented by orange bars. All configurations executed in a pseudo-parallel manner are situated in the left half of the diagram, while all configurations executed in a parallel manner are located on the right half. Each half comprises three principal groups, with the number of TpT varying from 1 to 10 to 100 from left to right. Each of these larger groups is composed of three smaller groups, in which the number of RpT is increased from left to right, from 1 to 10 to 100. In each of these smaller groups, the ratio of NCIt to CIt is varied in

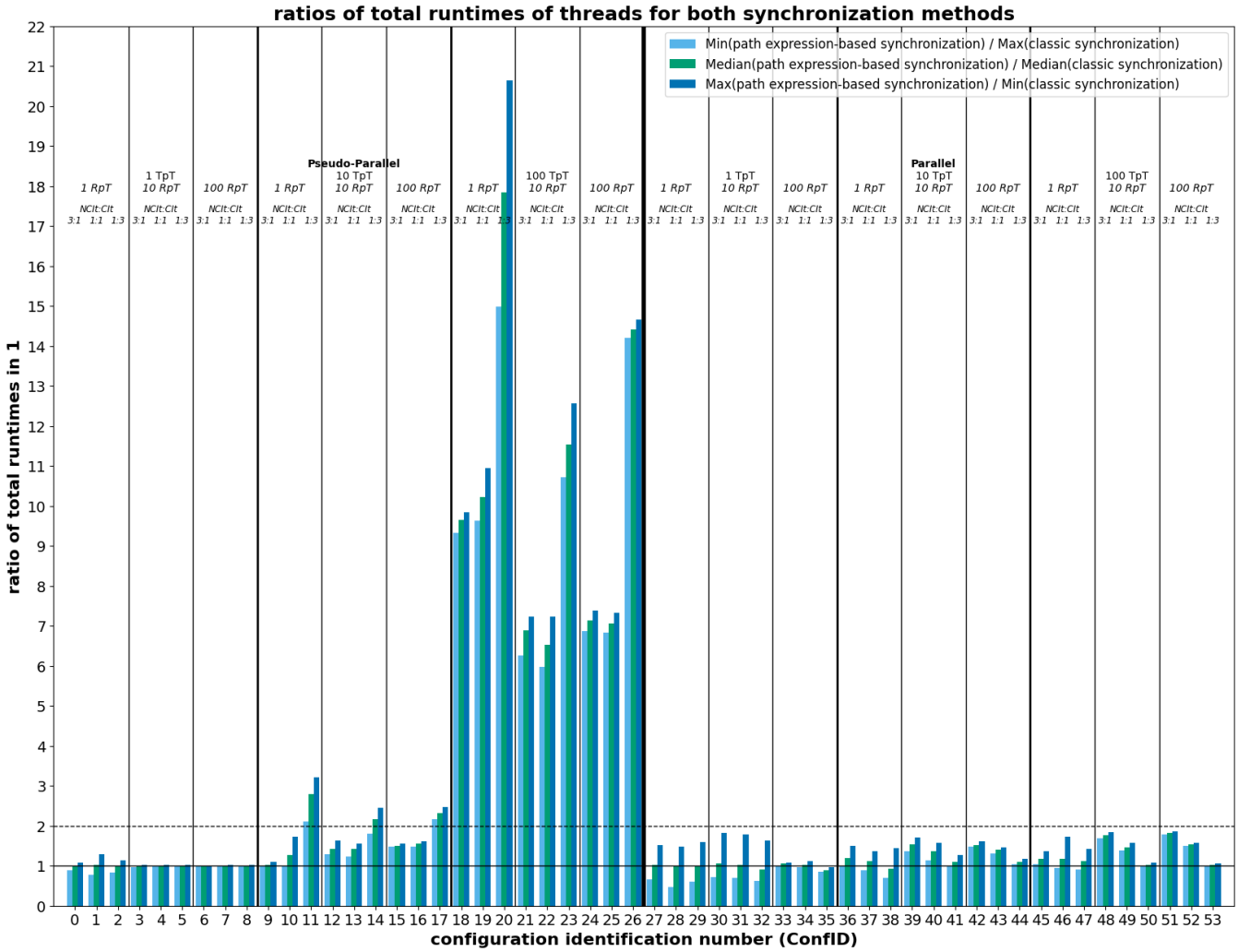


Figure 3: Ratios of total runtimes of all threads for both synchronization methods; plotted are the best, average and worst observed ratios of path expression-based synchronization. The values depend on the configuration (*ConfID*), which includes the execution mode {pseudo-parallel, parallel}, threads per type (TpT) {1, 10, 100}, and repetitions per thread (RpT) {1, 10, 100}. Ratio of non-critical to critical iterations (NCIt: CIt): 3:1, 1:1 and 1:3.

a stepwise manner, beginning with 3,000,000 NCIt:1,000,000 CIt, then decreasing to 2,000,000 NCIt:2,000,000 CIt, and finally reaching 1,000,000 NCIt:3,000,000 CIt.

The ratios of the running times for C and PE synchronization are shown in Figure 3. The different ratios of the running times are plotted on the ordinate. The various experiment configurations are plotted on the abscissa. The best, average and worst observed ratio from the perspective of PE synchronization is shown for each configuration from left to right in blue, green and deep blue respectively. The remaining structure of the illustration with regard to the various configurations is identical to that of Figure 2.

6 Discussion

As anticipated, the observable step-like increases in runtime align with expectations. Notably, the runtimes, particularly for C synchronization, consistently increase proportionally to the combined number of orders of magnitude, dependent on TpT and RpT. This phenomenon aligns with the premise that an increase in TpT and RpT by one order of magnitude equates to an equivalent rise in the operational load by two.

The observation that the runtimes for C and PE synchronization remain primarily unchanged when 1 TpT is applied but show significant differences when 10 or 100 TpT are utilized suggests that thread count may be the principal factor influencing the overhead of the PE approach. This indicates

that most observed runtime differences are likely attributable to active waiting, significantly impacting performance. Threads are awakened from the synchronizer queue but are not yet permitted to execute. They then monitor their status and wake other threads until, eventually, the scheduler awakens the thread that is ready to proceed.

This hypothesis is corroborated by the observation that the overhead of PE synchronization increases as the load ratio shifts into the critical section. This elevates the probability that the time slice of the actually productive thread will expire before its completion as the time spent in the critical section increases and active waiting takes place as a consequence.

The reduction in runtimes for PE synchronization when switching from pseudo-parallel to parallel execution mode by a factor of more than three provides further substantiation of this interpretation. Given that the number of CPU cores utilized in parallel execution mode is three, the theoretical speedup is constrained to a factor of three unless additional effects, such as the reduction of active waiting, are considered. The availability of multiple cores allows for a reduction in active waiting by approximately three. It also permits the scheduler to prioritize productive threads, identified by their high utilization and extensive use of their time slice, over unproductive waiting threads, which can significantly reduce active waiting.

With regard to the runtime ratio between the two synchronization methods, it can be stated that the runtime of PE synchronization in practical multi-core operation never deviates from the runtime of C synchronization by more than a factor of two. Indeed, there are configurations in which the ratio is nearly equal to one, such as for the high load scenario with ConfID 53. Given that classical synchronization should be a reasonable approximation of best-effort synchronization in terms of performance, this suggests a favorable prognosis regarding the tradeoff in using path expression-based synchronization. The performance benefits, such as the presumed greater ease of use, make the PE synchronization method a confident choice.

When it comes to expressive capacity, automaton-based path expressions are clearly superior to predicate-based ones. However, it's important to note that the approach of using two dummy procedures and their counters to dynamically generate any value at runtime has its own set of advantages and limitations. The gradual increase or decrease of the counters, as opposed to the immediate change with real variable references, is one such advantage. But there are also drawbacks, such as higher overhead and potential inaccessibility of the dummy processes. Extending predicates introduces new possibilities for more straightforward and generalizable solutions to synchronization problems.

In terms of user-friendliness, the prototype implemented in this work has already demonstrated significant enhancements compared to traditional synchronization techniques, as perceived by the authors. Implementing the considered synchronization scenario was more rapid and free of errors, but the construction of the corresponding automaton also facilitated a comprehensive understanding of the scenario's requirements. It should be noted that the specification of the path expression represents a fundamental prerequisite for synchronization, as even in instances where classical synchronization is employed, it is essential first to determine which sequences are permissible under which conditions. This is precisely the role that path expressions fulfill by design.

7 Conclusion

We presented the approach of automata-based path expressions, which represents a functional extension of the predicate-based path expressions proposed by Andler [1]. We developed a working prototype in C++ and evaluated its performance. The results demonstrated that for a complex synchronization example in a practical multi-core environment, the runtime difference between the path expression-based synchronization and the classical approach was consistently below a factor of two, with some configurations exhibiting a runtime nearly identical to that of the classical approach. Under the assumption that implementing synchronization via path expressions can be more straightforward, faster, and less susceptible to errors than the classical approach, these findings suggest a promising outlook for the potential benefits of path expression-based synchronization in modern applications.

Nevertheless, further research into the properties of path expression-based synchronization is warranted. Potential avenues for further investigation include optimizing the implementation to define the performance limits and directly mapping the specification given by an automaton-based path expression to synchronization primitives, as initially proposed by Campbell and Habermann [3].

To enhance user-friendliness, it would be advantageous to automate the utilization of path expression-based synchronization, beginning at the user's synchronization scenario specification stage. A suitable graphical interface with an integrated debugger could also optimize the creation of the specification using the automata.

Moreover, empirical studies employing a suitable prototype should corroborate our assessment of path expression-based synchronization's user-friendliness.

References

[1] Sten Andler. 1979. *Predicate path expressions.* In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 226–236. https://doi.org/10.1145/567752.567774

[2] Roy H. Campbell and Robert B. Kolstad. 1979. *Path expressions in Pascal.* 212–219. Int Conf on Software Eng, 4th, Proc Tech Univ of Munich ; Conference date: 17-09-1979 Through 19-09-1979.

[3] R. H. Campbell and A. N. Habermann. 1974. *The specification of process synchronization by path expressions.* In *Operating Systems*, E. Gelenbe and C. Kaiser (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 89–102.

[4] cppreference.com. [n. d.]. Type. https://en.cppreference.com/w/cpp/language/type

[5] L. Flon and A. N. Habermann. 1976. *Towards the construction of verifiable software systems. SIGMOD Rec.* 8, 2 (mar 1976), 141–148. https://doi.org/10.1145/984344.807132

[6] Gomez, Ernesto Soto. [n. d.]. PathEx. https://github.com/EStog/PathEx

[7] A. N. Habermann. 1975. *Path Expressions.* Department of Computer Science, Pittsburgh, PA, USA.

[8] P.E. Lauer and R.H. Campbell. 1975. *Formal semantics of a class of high level primitives for coordinating concurrent processes. Acta Informatica* 5 (1975), 297–332. https://doi.org/10.1007/BF00264564

[9] P.E. Lauer, P.R. Torrigiani, and M.W. Shields. 1979. *COSY a system specification language based on paths and processes. Acta Informatica* 12 (1979), 109–158. https://doi.org/10.1007/BF00266047

[10] Robert Love. 2010. *Linux Kernel Development* (3rd ed.). Addison-Wesley, Boston, MA, USA.

[11] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System* (2. ed. ed.). Addison-Wesley, Upper Saddle River, NJ.

[12] F. Schon, W. Schroder-Preikschat, O. Spinczyk, and U. Spinczyk. 2000. On interrupt-transparent synchronization in an embedded object-oriented operating system. In *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000) (Cat. No. PR00607)*. 270–277. https://doi.org/10.1109/ISORC.2000.839540

[13] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2010. *Operating System Concepts Essentials.* Wiley Publishing.

[14] Youfeng Wu and Ted G Lewis. 1989. *Parallelism encapsulation in C++.*