International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Nature-Inspired Optimization Based Multithread Scheduling For Program Segments

[1]Yogesh Sharma [2]Ajeet Kumar Vishwakarma, [3]Suneetha K, [4]Ashendra Kumar Saxena

Submitted:18/04/2023 **Revised:**11/06/2023 **Accepted:**22/06/2023

Abstract*:* The utilization of the processors, responsiveness, resource sharing, and efficient thread usage are the only benefits of multicore processors that allow multithreading techniques. Consequently, programming languages must allow multithreading programming to have these benefits. Since most ancient program codes were created sequentially, older software cannot work with this method. This was a difficult anytime multithreading code was being converted from old sequential procedures. There is a need for further optimization despite the availability of multiple multithreading algorithms due to discrepancies in their overhead, efficiency, and speedup. This demonstrates the efficacy of the optimization performed by a lightning search optimization over a wide range of problems. To develop multithreaded code while keeping a sequential one in mind, this research presents a linear simplex-elite integrated lightning search optimization (LS-EILSO) for multithreading scheduling. The number of stages of speedup execution time, efficiency, and cost of the LS-EILSO approach have all been evaluated. The most significant speed achieved by LS-EILSO with 32 threads is 11.98, while the average error rate between experimental and analytical cost numbers is 14.56 percent, as shown by experimental data. Additionally, it has been demonstrated that LS-EILSO can support more threads and achieve higher speedup when compared to intermediate representation based on LSO. As an illustration, the results reveal that when employing 32 threads both ways, LS-EILSO achieves a speedup of 11.71, about three times faster than the 3.77 speedups obtained by the current and planned.

Keywords: *Multithread scheduling, program segments, speedup multicore processors, linear simplex-elite integrated lightning search optimization (LS-EILSO)*

1. Introduction

In current society, Multithreaded programs are a significant focus of automated verification and bug detection techniques due to the prevalence of concurrency errors like deadlocks and data races (atomicity violations in general) that can arise from subtle interactions between threads not anticipated by the developers. Methods that exhaustively explore the space of all possible thread interleaving at runtime are best suited for accurately detecting potential concurrency issues. The state space explosion prevents detailed systematic verification from applying to large real-world multithreaded software systems, even though numerous approaches and tools in this area currently exist. The large number of thread interleaving that must be examined, even for relatively modest multithreaded programs, is the primary cause of poor scalability. Modern approaches and tools partially tackle this problem by employing various algorithmic enhancements, optimizations, and heuristics, such as partial order reduction variants, modular thread verification, iterative context-bounded verification, and symbolic predictive analysis based on dynamic event recording. However, validating large and complex software systems is still challenging [1]. Predictability is a crucial feature of modern real-time embedded systems. The complexity of today's electronics continues to rise. Due to intense rivalry, most suppliers now provide hardware capable of supporting cutting-edge features. However, the complexity of electronic embedded systems' architectures can undermine their predictability, and engineers must invest countless hours into the design process to ensure that the final product is reliable. Researchers worldwide have put in a lot of time and energy over the past decade trying to design architectures that guarantee timed repeatability. Real-time systems have various uses, all requiring complete predictability [2]. Due to the exponential number of thread interleaving, multithreaded program analysis is infamously tricky. Multithreaded programming may be flawed owing to memory mistakes and assertion violations not caused by race circumstances, even though race detectors can help developers uncover and repair such flaws before the code

1Professor, School of Engineering & Technology, Jaipur National University, Jaipur, india, Email Id- yogeshchandra.sharma@jnujaipur.ac.in

2Assistant Professor, School of Engineering and Computer, Dev Bhoomi Uttarakhand University, Uttarakhand, India, Email Id: socse.ajeet@dbuu.ac.in

3Professor, Department of Computer Science and IT, Jain(Deemed-to-be University), Bangalore-27, India, Email Id: k.suneetha@jainuniversity.ac.in

4Professor, College of Computing Science and Information Technology, Teerthanker Mahaveer University, Moradabad, Uttar Pradesh, India, Email id: ashendrasaxena@gmail.com

is deployed.

Moreover, programmers may need more evidence before being convinced of the dangers of race condition warnings. The stability and security of multithreaded software have been the subject of recent attempts to close this gap. By studying dependencies in the code's state space, these methods infer schedules that identify bugs in multithreaded programs. It believes a generic property-directed technique is required for the analysis of multithreaded programs. Memory safety and other custom-defined guarantees are among the qualities of interest. Analysis should correctly identify property-relevant data-flow dependencies so that the vast state space of scheduling scenarios may be searched efficiently [3]. We consequently presented the LS-EILSO-based multithread scheduling for program segments.

2. Related Work

The research [4] presented PyOMP, a solution that makes Python compatible with OpenMP. Python code written using OpenMP by programmers is generated by Numba, compiles to LLVM, and runs with performance comparable to C code written with OpenMP. The research [5] offered an original pairwise-based algorithm for incremental verification of multithreaded programs, whose central notion is a systematic investigation of all potential thread interleavings only for specific relevant pairs of threads. An architecture template for autonomous accelerator development for graph analytics and irregular applications is presented in this study [6]. The research [7] provided deals with the issue of synchronizing internal program threads with processes that are operating on remote machines. Using a queue-based method for thread synchronization with weaker operation semantics is suggested. Study [8] proposed a plan for accelerating the discovery of variables shared by several threads and, consequently, the detection of concurrency issues. This method makes use of a static scheduling scheme. The research [9] presented an effective event-based problem-scheduling plan and a new multithreading in-order pipeline microarchitecture design for RISC-V. A study [10] demonstrated that the latter technique needs deterministic scheduler configurations dynamically adjusted to the current application load during runtime to achieve meaningful performance benefits. According to the research [11] is a straightforward technique for quickly implementing simultaneous multithreading (SMT) in complex, prioritized real-time systems. An earliest-deadline-first (EDF) scheduler that uses SMT is created by combining integer linear programming and heuristic bin-packing. Developers worry about multithreaded apps' inconsistent latency.

Code errors, poor database architecture, thread imbalances, resource congestion, and system saturation can create performance issues. The initial problem in complicated systems like the Chromium browser, our focus in this study [12], is collecting precise unified information from multiple layers. The research [13] discussed planning a multi-skilled workforce to build a robust maintenance service network for high-value assets. By maximizing the capacity of the repair shop staff and achieving workforce heterogeneity through cross-training, they increase the effectiveness of the maintenance network. Study [14] presented an integrated thread- and data-mapping framework for NoC-based Scratchpad Memories (SPMs) many-cores when running multithreaded multi-phase applications. The research [15] offered a unique multithreaded implementation of the D-bar approach that interfaces C code that uses the threads package with a front-end MATLAB/Octave program. A study [16] offered a cybernetic control technology that may be used for sophisticated software systems. According to this method, cybernetic control objects control the software systems, and specialized meta-programming platforms create the class libraries specifying the many sorts of these cybernetic control objects.

The remaining sections of this research are as follows: Part 2 contains the related works; the proposed methodology is introduced in Part 3; the result and discussion of the study are in Part 4; the conclusion is in Part 5.

3. Methodology

Even though standard LSA converges quickly, it has limitations in other areas, including solution accuracy, the ability to address multimodal optimization issues, and avoiding premature convergence. To compensate for LS drawbacks, we augment it with two additional optimization strategies: the EILSO.

The LSO takes its name and inspiration from the illuminating natural occurrence of lightning, whose discharges have both probabilistic and sinuous qualities during a rainstorm. This optimization approach builds on the concept of step leader propagation, which is itself a generalization of the idea of projectiles. The optimization particle stands in for the current population and is like the projectile in a war game. The energy at the current step's leading edge is the LSO solution. Lead shot, space shot, and transition shot are the three types of projectiles used in LSO. The projectiles in transition generate a population of potential solution leaders in the first stage, the projectiles in space undertake exploratory missions to unseat the current leader, and the projectiles in the lead search for and exploit the best solution.

A. Transition Projectile

The transition projectile $O^T = O_1^T, O_2^T \ldots., O_m^T$ is launched in a direction chosen at random from the thunder cell during the initial stages of the creation of a stepped leader. As a result, we can treat it as if it were a number picked at random from a uniform distribution:

$$e(w^S) = \begin{cases} \frac{1}{a-b} for\ a\ \leq w^S \leq a \\ 0 \qquad elsewhere, \end{cases} \quad (1)$$

The solution space is bounded by a minimum of a maximum, where w^S is a random value that may produce a solution. It can be shown that $LS = ls_1, ls_2, \ldots\ldots, ls_m$ for a population with m stepped leaders; the solution dimension necessitates m random projectiles.

B. Space missile

It is possible to describe the step + 1 location of the space projectile $O^T = O_1^T, O_2^T \ldots., O_m^T$ based on a random draw from a parametric proportional distribution.

$$e(w^t) = \begin{cases} \frac{1}{\mu} e^{-w^t/\mu}\ for\ w^t > 0 \\ 0 for\ w^t > 0 \end{cases} \quad (2)$$

The resulting expression for O_j^T position and angle at step + 1 is the distance between the lead projectile O^T can be calculated using the formula.

$$O_{j\_new}^T = O_j^T \pm \exp rand(\mu_j) \quad (3)$$

At step + 1, μ_j, the projectile's kinetic energy will be adjusted to become the new step leader's kinetic energy if the projectile's $O_{j\_new}^T$ is greater than O_j^T. Until the next stage, *exp*ansion, the modifications will not take effect.

C. Leading Missile

The descent of the lead projectile, PL, can be represented by a normal distribution random number with the following form:

$$e(w^K) = \frac{1}{\sigma\sqrt{2\pi}} f^{-\frac{\left(w^K-\mu\right)^2}{2\sigma^2}} \quad (4)$$

The shape parameter (μ_K) determines how far the randomly generated lead missile can travel before it returns to its original place. The scale parameter (σ_K) gives this missile a degree of exploitability. The value of the scale parameter K decreases exponentially with increasing distance from Earth or approaching the best solution. So, we can write down the expression for w^K in step +1 stage as:

$$O_{new}^K = O^K + normrand(\mu_K, \sigma_K) \quad (5)$$

$Normand$ is a random number with the same chance of occurrence as any other number selected from the normal distribution. In cases where μ_K is greater than σ_K is likewise changed at step + 1 to O^K new. Any changes won't take effect until the subsequent phase.

D. Creating a technique

Forking, or the appearance of two parallel and symmetrical branches, is a crucial feature of a stepped leader. The proposed method incorporates two different forms of creation. Initially, as shown in the following diagram, symmetrical channels are created when the projectile's nuclei hit with an inverse number.

$$\overline{O_j} = b + a - O_j \quad (6)$$

Here b and a are the limits of the system $\overline{O_j}$ and O_j the projectiles in issue and their opposites, respectively. The forking leader chooses $\overline{O_j}$ or O_j with a higher fitness value to ensure the population's continued existence. In the second form of forking, the most failed leader's energy is redistributed among many propagation attempts, creating a channel at the leader's successful step tip. The burden of leading the unsuccessful pack can be distributed by establishing a maximum allowed number of attempts as channel time.

E. Linear Simplex

When speed and accuracy matter most in local search, a straightforward approach has several advantages. To find the local minimum of a function, Nelder and Mead suggest a simple line search strategy that doesn't use derivatives. We select the O worst-performing step leaders and apply LS to optimize their locations. This enhances the algorithm's proximity to optimality, exploitability, and speed at reaching optimality. The LS methods used in this analysis are outlined below.

Step 1: Assuming that all of the step leaders have been explored, determine the best (wh) point, the worst (wa) point, and the K-worst wx point using the values of the objective function.

Step 2: Find the midpoint between the ideal solution (wh) and the worst-case scenario (w).

$$\overline{w} = \frac{wh+wa}{2} \quad (7)$$

Step 3: The point wq is reached by reflecting wx via the center $\overline{w}$. Now would be an excellent time to compute the value of the objective function.

$$wq = \overline{w} + \alpha(\overline{w} - wx) \quad (8)$$

The coefficient of reflection is 1, thus.

Step 4: Perform expansion to produce a new point wx and if $e(wq) > e(wh)$; otherwise, proceed to Step 5. Replace wx with wx if $e(wx) < e(wh)$; otherwise, replace wq with wh.

$$wx = \bar{w} + \gamma(wq - \bar{w}) \quad (9)$$

The expansion coefficient is defined as 1.5.

Step 5: if reflection fails ($e(wq) > e(wx)$), if point xw can be shrunk to form point xc, we do so; otherwise, we proceed to step 6. If and only if $e(wd) = e(wx)$, replace.

$$wd = \bar{w} + \beta(wx - \bar{w}) \quad (10)$$

The contracting ratio is = 0.5.

Step 6: The point wx is reduced in size to produce a new point wt if $e(wq) < e(wh) < e(wx)$. The contraction coefficient is the same as the shrinkage coefficient.

$$wt = \bar{w} + \beta(\bar{w} - wx) \quad (11)$$

At each iteration, the LS helps the current worst-step leaders improve their position to a better state than the ideal one. This improves the algorithm's convergence accuracy and rate by preventing the population from exploring suboptimal solutions instead directing it toward the global optimum.

F. EILSO

While LS enhances the EILSA convergence accuracy, getting stuck in a local optimum is still relatively simple. Thus, the EILSO approach was developed to combat this problem. Tizhoosh publicly set a novel model of machine intelligence called opposition-based learning (OBL), which considers both the current and its opposite estimates simultaneously to arrive at a superior solution. It has been demonstrated that, compared to a randomly chosen candidate solution, an inverse candidate solution is more likely to be closer to the global optimal solution. EILSO employs the opposition-based population generation found in OBL to create elite step leaders to increase the genetic variation among LSA populations. W_i, $W_{j,2}$,..., $W_{j,i}$ = ($W_{j,1}$, $W_{j,2}$,..., $W_{j,i}$) is the fitness value for the top-tier tempo setter/step leader. W_j = ($W_{j,1}$, $W_{j,2}$,..., $W_{j,}$ i, C) is an alternate step-leader solution. The same answer is obtained by writing $w'_{j,i}$= ($W_{f,1}$, $W_{f,2}$,..., $W_{f,}$ i, C) as $w'_{j,i}$ = $w'_{j,1}$,..., $w'_{j,2}$= ($w'_{j,1}$, $w'_{j,2}$,..., $W_{j,}$ C).

$$w'_{j,i} = l \cdot (Ka_i + Va_i) - W_{f,i}, j = 1,2, \dots, n; j = 1,2, \dots, C \quad (12)$$

The search bound is (Ka_i, Va_i) if the population size is n, w has C dimensions, and l is between 0 and 1. The following formula is applied to the data to determine if the elite, based on resistance step leader $w'_{j,i}$ is beyond the search region:

$$w'_{j,i} = rand(Ka_i, Va_i), if\ w'_{j,i} < Ka_i\ or\ w'_{j,i} > Va_i \quad (13)$$

Following these guidelines, the following actions constitute EILSO-OBL.

Step 1: First, we use (12) to generate the most suited step leader Xe possible. This will be the foundation for our n-person elite solid opposition.

Step 2: When the leading resistance organization's step leader vanishes outside the search radius, we apply (13) as the heuristic.

Step 3: 2n step leaders compete for entry into the next generation, and only those with the highest fitness scores will survive.

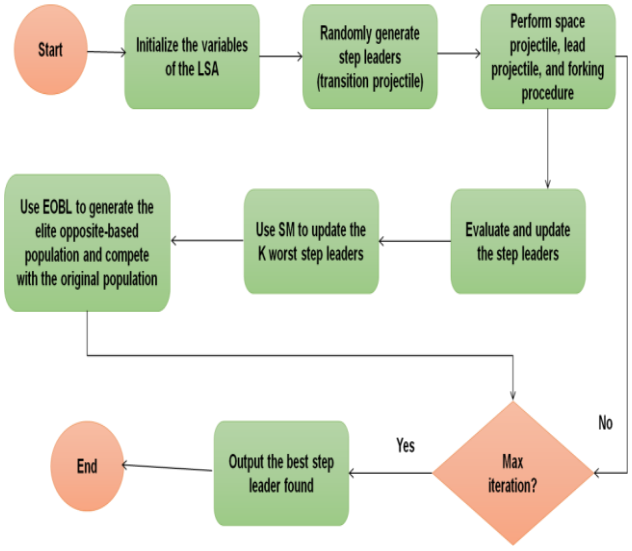

Fig.1. Flowchart of the LS-EILSO

The algorithm's exploitation and exploration are optimized by including the LS and EILSO techniques, as demonstrated in Fig.1 flowchart of the EILSO-LS flowchart. Taking into account the maximum number of iterations, population size (m), problem dimension (C), and objective function cost (Cof), the findings above indicate that LSA-ELSO has a space complexity of O(2m C) and a time complexity of O.

4. Result and Discussion

In this section, the proposed method is efficacy to compared that of previously used approaches such as reinforced manta rayforaging (RMRFO), extreme gradient boost (XG-boost), and grey-box fuzzing teqchniques (G-BFT). Speed-up, efficiency, cost and execution time were important metrics studied using proposed and current approaches.

We detail the experimental setup, discussing the hardware, software, and tools utilized to perform LS-EILSO

approaches and the benchmark datasets employed throughout the study [17]. The following infrastructure is used in this work as an experimental platform a multicore system consisting of "Intel(R) Core(TM) i7-8550U 1.80 GHz CPU processors, 8 CPUs of cache memory per processor, and 16 GB RAM". The OS that was used was “Ubuntu 17.10 64-bit”. The C# programming language used in Microsoft's Visual Studio 2017 IDE was also used to implement the LS-EILSO approaches. We also applied our findings to the “dense matrix-vector product DenseAMUX” benchmark. To show the speedup made feasible by the LS-EILSO approach, we also used the code from Fig. 2.

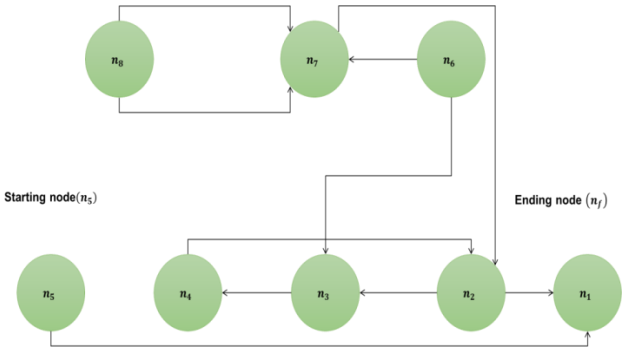

Fig.2. Modula-like programming flow-graph

This vector is valuable in many applications. A sequential program can take a long time to solve a problem with a large input size. Matrix-vector products take time to execute sequentially. Since the execution time must be smaller than O, a multithreading approach is needed to solve the problem, mainly if Q is a significant number. The following studies account for the known influences on speedup and efficiencies, such as matrix size, thread size, and thread count. Matrix sizes are employed, with 5000*5000 being used and labeled M5000; these datasets originated from the “DenseAMUX” benchmark.

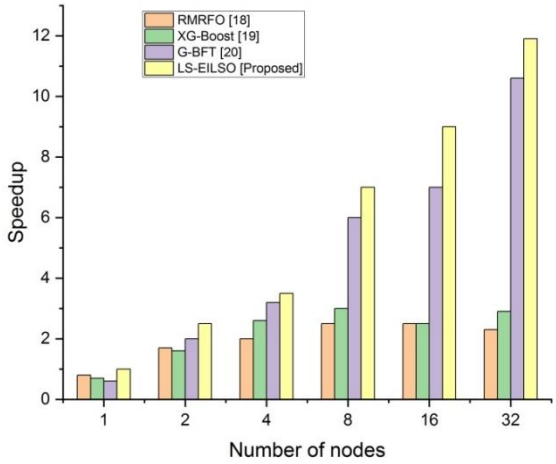

Fig.3. Comparison of the speed

up and the relative rates of speedup is shown in Fig. 3. However, it is well-known that the number of concurrent threads and their performance are affected by the thread size. The data shows speedup values are low when using smaller sizes and high when using larger ones. The experiments are run with 1, 2, 4, 8, 16, and 32 threads, with each thread having a size of L, equal to 80,000, as defined in three existing methods (RMRFO, XG Boost, G-BFT) and suggested (LS-ELSO). Each thread will receive a maximum of 80,000 unscheduled nodes from SM on each cycle.

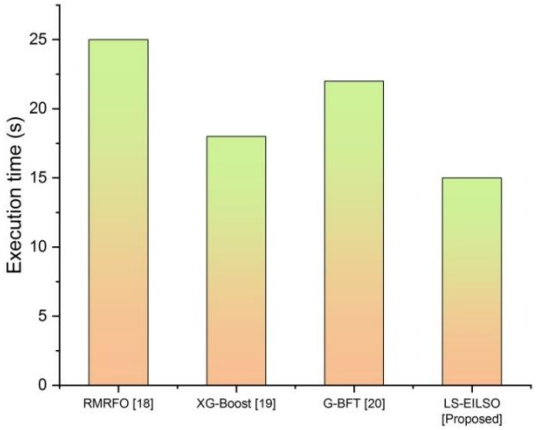

Fig.4. Comparison of execution time

A process's execution time is the amount of time it actively makes use of computer resources. It's conceivable that varying amounts of time will be needed to finish different tasks from the same assignment. The time it takes for a program or a piece of code to execute from when it is started until it is finished is known as its execution time, runtime, or elapsed time. It is an important parameter to measure the efficacy and effectiveness of multithreaded programming methods. Fig.4 shows the comparison of execution time. As a result, the proposed work LS-EILSO has the lowest execution time than the existing works RMRFO, XG Boost, and G-BFT.

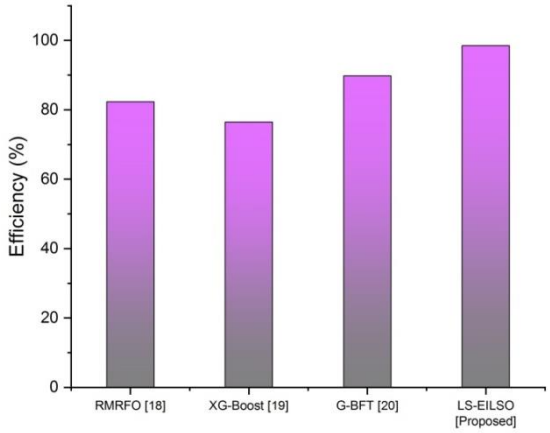

Fig.5. Comparison of the efficiency

Fig.5 depicts the comparison of the efficiency. The efficiency of a multithread scheduling method is measured by how well it makes use of available system resources and how well it meets performance goals. The needs and features of the application should inform the decision of

the scheduling algorithm to use. A scheduling strategy's effectiveness is measured by how well it enhances the running application in terms of resource consumption, responsiveness, and overall system performance. Thus, LS-EILSO has the fastest decryption time of RMRFO, XG-Boost, and G-BFT.

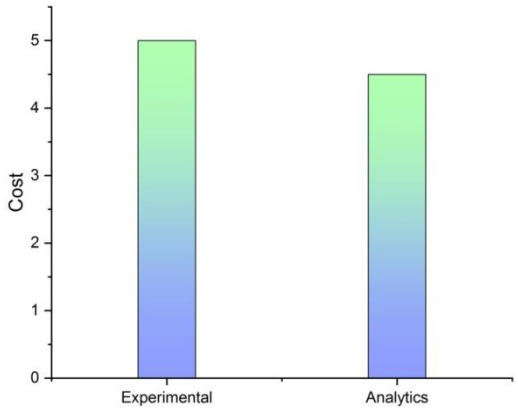

Fig.6. Comparison of the experimental and analytic cost

Fig. 6 depicts the comparison of the experimental and analytic costs. It is usual to use both practical and analytical methods to determine how much a specific multithread scheduling algorithm costs and how well it performs. The cost of multithread scheduling algorithms can be better understood by academics and developers when experimental and analytic studies are combined. Validation in the actual world is provided via exploratory analysis, while insights and predictions can be made by analytical research in a laboratory setting. Both approaches have advantages and disadvantages, and they can work together to provide a more thorough analysis of multithread planning techniques. The suggested experimental work has the highest cost of analytics.

5. Conclusion

In This research introduced the LS-EILSO technique, a novel multithreaded scheduling approach for software components. The lightning search optimization known as LS-EILSO distributes the parallel versions of sequential programs over several threads. The LS-EILSO approach consists of two steps. To classify the nodes and identify their counterparts, the suggested method is initially used by LS-EILSO to analyze the sequential program's flowchart. The second phase uses the execution paths created by LS-EILSO. Second, a chemical reaction optimizer is used to assign threads to their execution routes progressively. As a final deliverable, LS-EILSO generates an incremental thread schedule that has been optimized. The outcomes were evaluated based on their performance against industry norms and guidelines. Many different criteria, both experimental and analytical, were used to assess the efficacy of the LS-EILSO method, and these evaluations were carried out on many different data sets. LS-EILSO was found to be successful 98.5% of the time and to be carried out in about 15 seconds. The testing results reveal that the best speedup occurs when LS-EILSO uses 32 threads on matrix M5000. Additionally, the LS-EILSO approach's experimental cost increases by a factor of fE where 1.43 B fEB 3.53. A growing analytical cost of fA where 1.45 B fAB 3.47 has also been seen for the LS-EILSO method. When all the numbers were added up, the results showed that the experimental costs were, on average, 14.56% higher than the analytical costs.

References

[1] Parízek P, Kliber F. Incremental Verification of Multithreaded Programs by Checking Interleavings for Pairs of Threads. Technical report; 2022 Jul 25.

[2] Antolak E, Pułka A. Energy-efficient task scheduling in design of multithread time predictable real-time systems. IEEE Access. 2021 Aug 30;9:121111-27.

[3] Yavuz T. SIFT: A Tool for Property Directed Symbolic Execution of Multithreaded Software. In2022 IEEE Conference on Software Testing, Verification and Validation (ICST) 2022 Apr 4 (pp. 433-443). IEEE.

[4] Soueidi C, El-Hokayem A, Falcone Y. Opportunistic monitoring of multithreaded programs. InFASE 2023 Apr 20 (pp. 173-194).

[5] Parízek P, Kliber F. Checking Just Pairs of Threads for Efficient and Scalable Incremental Verification of Multithreaded Programs. ACM SIGSOFT Software Engineering Notes. 2023 Jan 17;48(1):27-31.

[6] Minutoli M, Castellana VG, Saporetti N, Devecchi S, Lattuada M, Fezzardi P, Tumeo A, Ferrandi F. Svelto: High-level synthesis of multi-threaded accelerators for graph analytics. IEEE Transactions on Computers. 2021 Feb 8;71(3):520-33.

[7] Thanagaraju, V. ., & Nagarajan, K. K. . (2023). A Detailed Analysis of Air Pollution Monitoring System and Prediction Using Machine Learning Methods. International Journal on Recent and Innovation Trends in Computing and Communication, 11(2s), 51–58. https://doi.org/10.17762/ijritcc.v11i2s.6028

[8] Tabakov AV, Paznikov AA. Using relaxed concurrent data structures for contention minimization in multithreaded MPI programs. InJournal of Physics: Conference Series 2019 Dec 1 (Vol. 1399, No. 3, p. 033037). IOP Publishing.

[9] Jahić J, Kumar V, Jung M, Wirrer G, Wehn N, Kuhn T. Rapid identification of shared memory in multithreaded embedded systems with static scheduling. InProceedings of the 48th International Conference on Parallel Processing: Workshops 2019

Aug 5 (pp. 1-8).

[10] Eni Y, Greenberg S, Ben-Shimol Y. Efficient Hint-Based Event (EHE) Issue Scheduling for Hardware Multithreaded RISC-V Pipeline. IEEE Transactions on Circuits and Systems I: Regular Papers. 2021 Oct 12;69(2):735-45.

[11] Habiger G, Hauck FJ, Reiser HP, Köstler J. Self-optimising application-agnostic multithreading for replicated state machines. In2020 International Symposium on Reliable Distributed Systems (SRDS) 2020 Sep 21 (pp. 165-174). IEEE.

[12] Osborne SH, Ahmed S, Nandi S, Anderson JH. Exploiting simultaneous multithreading in priority-driven hard real-time systems. In2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) 2020 Aug 19 (pp. 1-10). IEEE.

[13] Rezazadeh M, Ezzati-Jivan N, Azhari SV, Dagenais MR. Performance evaluation of complex multi-thread applications through execution path analysis. Performance Evaluation. 2022 Jun 1;155:102289.

[14] Turan HH, Kosanoglu F, Atmis M. A multi-skilled workforce optimisation in maintenance logistics networks by multi-thread simulated annealing algorithms. International Journal of Production Research. 2021 May 3;59(9):2624-46.

[15] Venkataramani V, Pathania A, Mitra T. Unified thread-and data-mapping for multi-threaded multi-phase applications on SPM many-cores. In2020 Design, Automation & Test in Europe Conference & Exhibition (DATE) 2020 Mar 9 (pp. 1496-1501). IEEE.

[16] Alsaker M, Mueller JL, Stahel A. A multithreaded real-time solution for 2D EIT reconstruction with the D-bar algorithm. Journal of Computational Science. 2023 Mar 1;67:101967.

[17] Dhablia, A. (2021). Integrated Sentimental Analysis with Machine Learning Model to Evaluate the Review of Viewers. Machine Learning Applications in Engineering Education and Management, 1(2), 07–12. Retrieved from http://yashikajournals.com/index.php/mlaeem/article/view/12

[18] Bozkurt EM. The usage of cybernetic in complex software systems and its application to the deterministic multithreading. Concurrency and Computation: Practice and Experience. 2022 Dec 25;34(28):e7375.

[19] Mahafzah BA, Jabri R, Murad O. Multithreaded scheduling for program segments based on chemical reaction optimizer. Soft Computing. 2021 Feb;25:2741-66.

[20] Malave SH, Shinde SK. Reinforced Manta Ray Foraging Optimiser for Determining the Optimal Number of Threads in Multithreaded Applications. International Journal of Intelligent Systems and Applications in Engineering. 2022 Dec 27;10(3s):17-26.

[21] Chen H, Guo S, Xue Y, Sui Y, Zhang C, Li Y, Wang H, Liu Y. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. arXiv preprint arXiv:2007.15943. 2020 Jul 31.

[22] Sun J, Shan L, Shu X. XGBoost Dynamic Detection for Data Race in Multithreaded Programs. InAdvances in Natural Computation, Fuzzy Systems and Knowledge Discovery: Proceedings of the ICNC-FSKD 2021 17 2022 (pp. 1251-1258). Springer International Publishing.