2024 IEEE 18th International Symposium on Applied Computational Intelligence and Informatics (SACI) | 979-8-3503-2952-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/SACI60582.2024.10619722

Java single vs. platform vs. virtual threads runtime performance assessment in the context of key class detection

Ciprian-Bogdan Chirila
Computer and Information Technology Department
University Politehnica
Timişoara, Romania
chirila@cs.upt.ro

Ioana Sora
Computer and Information Technology Department
University Politehnica
Timişoara, Romania
ioana@cs.upt.ro

***Abstract*—Key classes are considered the most important classes of a software system. They represent the starting point for reengineering or documentation processes. The detection of key classes is considered crucial in the state of the art, many studies are focused on automatic detection based on class graph system representation.**

Studies show that class attributes computed with algorithms like Hyperlink-Introduced Topic Search (HITS) and PageRank (PR) give the best precision and recall performance values in detecting key classes. The runtime execution of the two algorithms is critical when they run on graphs having different class relationship weights. To ameliorate the time execution problem we experiment with the two algorithm parallel implementations based on Java: i) single thread, ii) platform or operating system threads and iii) virtual threads. The experiments are fulfilled on a set of 14 Java projects.

The results show that single thread implementations for project having a relative small number of classes, namely under 1,200, perform better than platform threads implementations. Conversely, virtual threads perform better than any single thread implementation. We conclude that virtual thread model speeds up the computation of attributes with a runtime decrease of 58.41% against the single thread model.

***Index Terms*—Java threads; Java virtual threads; Hits Algorithm; PageRank algorithm; key classes detection**

I. Introduction

In this paper we present a use case for Java threads in the context of key class detection framework. Concurrency in Java is attained via the Java Thread API, which acts as a platform thread encapsulating an underlying OS thread, enabling Java applications to execute concurrent code.

With the advent of Loom project [1] initiated by Oracle we can benefit from lightweight, user-mode threads, also known as fibers, to the Java Virtual Machine (JVM). These fibers are designed to be more efficient and scalable than traditional OS threads, allowing Java developers to handle massive numbers of concurrent tasks with minimal overhead.

Parallelization is the technique of dividing a large computational task into smaller sub-tasks that can be executed concurrently on multiple processors or cores, with the goal of reducing overall computation time.

Fibers are lightweight, user-mode threads that can be scheduled by the JVM without relying on the operating system's thread scheduling. They are cheaper to create and manage compared to OS threads, making it possible to have a much larger number of concurrent tasks.

Virtual threads are a specific type of fiber introduced in Loom. They are designed to be with minimal memory overhead, and can be created and scheduled quickly by the JVM. Virtual threads made their debut as a preview API in JEP 425 and were included as a preview feature in Java Development Kit (JDK) 19. They are characterized by their lightweight nature and the ability to execute Java code on an underlying OS thread without monopolizing it for the entirety of the code's execution. This allows multiple virtual threads to share the same OS thread, significantly increasing the number of virtual threads that can be instantiated compared to traditional platform threads. Leveraging virtual threads instead of conventional threads in Java web services may facilitate the processing of thousands of requests without the necessity of creating additional operating system threads.

Loom leverages continuations to implement fibers. Continuations allow the state of a computation to be captured and resumed later, enabling efficient context switching between fibers. It aims to simplify asynchronous programming in Java by providing constructs like "async" and "await", similar to those found in languages like JavaScript and C#. These constructs allow developers to write asynchronous code in a more sequential and intuitive manner. Loom is designed to be backward compatible with existing Java code. Developers can start using fibers in their applications without making significant changes to their code.

In Figure 1 we present conceptually our approach. We start from a set of 14 analyzed systems written in Java that we will call projects. These projects are parsed by a parsing component (Parser) analyzing the class relationships and building the weighted graph. The software system is modeled as a class graph where the classes/interfaces are considered nodes and the class relationships are considered edges. In some studies several class relationships are considered (e.g. inheritance,

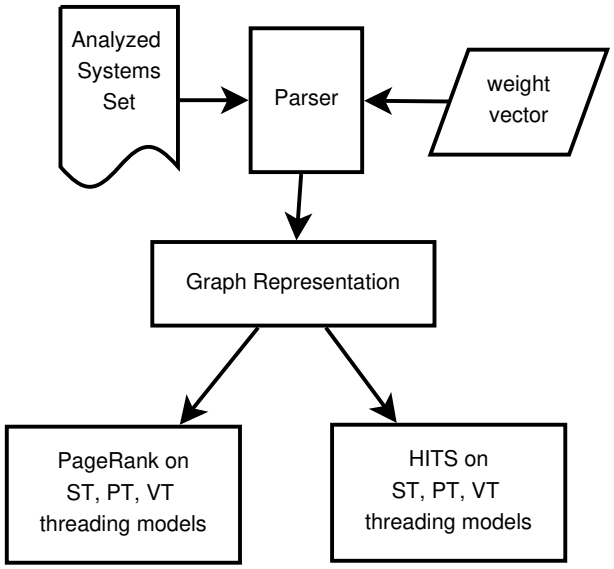

Fig. 1. Runtime Comparison Approach

cast, method call etc.) and they are are assigned different weights [2].

We compute for each class two attribute values provided by two algorithms HITS [3] and PageRank [4]. Previous research indicate that these attributes are the best for automatic detection of key classes [5], [6]. The weights for the graph edges are obtained from a Weight Vector component. This component may be connected to weight exploration algorithms based on enumeration or heuristic methods. When a large number of weight vectors (e.g. 30,000 vectors) are applied on the graph, the attribute computation algorithms have to run multiple times, once on each weighted graph [7]. In this context the execution time becomes critical. The problem is even more severe when the number of classes of an analyzed system is large. On a large system, of over 4,000 classes, the execution time for one algorithm may take up to 90 seconds. The considered analyzed systems size vary from few hundred of classes to few thousands of classes.

The detection of key classes is based on running the two algorithms (HITS and PageRank) to determine class attribute importance values. Classes are ranked in descending order of their importance and thus, the top k classes are considered to be the key classes, where k=20 or 30 or 50.

Fortunately, the two algorithms HITS and PageRank allow straightforward parallelization. On the weighted graph representation we run the two algorithms using three threading models: i) with the use of a single thread (ST) - one threaded application; ii) with multiple platform threads (PT) - present in early Java versions; iii) with multiple virtual threads (VT) - enabled by project Loom.

We consider running the threads on two platforms MS Windows [8] and Ubuntu Linux [9]. For these six algorithm implementations (2 attributes x 3 threading models) running on four machines and on 14 projects we extract the execution times and we store them in CSV tables. We are not interested in the implementations results, but in their execution times. Finally, the execution times are compared, analyzed and several conclusion are drawn about the overall performance of threading models in this context.

In this paper we answer the following research questions: (RQ1) How is the runtime performance of ST vs. PT models on each machine and project? (RQ2) How is the runtime performance of ST vs. VT models on each machine and project? (RQ3) What are the runtimes of the whole set of projects for each threading model ST, PT, VT? (RQ4) Which are the projects which consume most of the resources when computing attribute values? (RQ5) Is there a significant time performance difference between the four machines? (RQ6) How can be clustered the set of analyzed projects, using different threading models for each cluster, to minimize the overall runtime?

The paper is structured as follows. Section II presents related works in the fields of execution threads and key class detection. Section III presents the design of the experiment. Section IV presents the results obtained from the experiment. Section V analyzes the experimental results. Section VI concludes and sets the future work.

II. Related Works

The work of [10] discusses challenges in using Java for parallel programming, highlighting issues like high threading overhead and performance degradation in multi-threaded applications on parallel systems. It introduces an analysis environment correlating various execution levels to understand these issues better. Additionally, it proposes scheduling mechanisms and policies to efficiently execute multi-threaded Java applications on multiprocessor systems, emphasizing bi-directional communication between applications and the execution environment for resource management. The concept of self-adaptive applications, adjusting their behavior based on resource allocation information, is presented, with examples in HPC and e-business environments. Evaluation results indicate improved performance and resource utilization through dynamic resource provision and application adaptation. In our context the threads will be mapped to the nodes of the graphs. The computation enabled by each node is independent of the rest.

In [11] is presented how to use virtual threads in developing web services. To build a high-performance web service in Java, Reactive WebFlux stood as the sole option. However, with the advent of virtual threads in Java 19, it is now being considered as a viable alternative. This study investigates the performance disparity among Spring applications utilizing conventional threads, virtual threads, and the reactive approach of Reactive WebFlux. The experimentation involved creating three prototypes, each subjected to calling an endpoint with a predetermined delay time while incrementing the number of requests per second until system failure. Results indicate that the prototypes employing virtual threads exhibited marginally superior performance compared to the reactive prototype. Further exploration is warranted to ascertain whether the

choice of the most optimized web server for the Reactive WebFlux application yields different outcomes, along with prospective inquiries into the performance of virtual threads when integrated with database operations. We use the same approach of comparing different threading models in a key class detection framework.

Numerous case studies have been conducted to analyze the performance disparities between traditional OS threads and Java's innovative virtual threads.

The work of [12] discusses the challenges posed by blocking operations, such as input/output (IO) operations, in concurrent applications due to their impact on efficiency and resource allocation. Traditional multithreaded applications face difficulties in managing these operations efficiently because of the single-thread access policy and reliance on heavyweight OS kernel threads. To address these challenges, the paper explores various asynchronous programming techniques and proposes a model of structured concurrency in the Java Virtual Machine (JVM) using virtual threads. These virtual threads, along with their accompanying schedulers, aim to provide a lightweight alternative to traditional OS threads, decoupled from OS resources and managed within the JVM environment. The research presented in the paper evaluates the efficiency of these schedulers for virtual threads in handling blocking operations, comparing them with traditional OS thread schedulers. It also discusses possibilities for reducing memory footprints and context switching costs associated with virtual threads.

In the work of [13], four HTTP server prototypes were developed to assess the number of OS threads initiated and the heap utilization in Megabytes for each prototype after every test iteration. Each test operation involved creating an object, writing it to a file, and then returning it as a response. Conventional Java and Kotlin threads initiated approximately 100,000 threads and consumed around 300-400 Megabytes of heap. Kotlin coroutines, on the other hand, started approximately 23 OS threads and utilized 52-99 Megabytes of heap. Java virtual threads commenced around 35 OS threads and utilized 16-64 Megabytes of heap. The study suggests that structured concurrent approaches like Kotlin and Java Virtual Threads exhibited notable performance improvements compared to their conventional counterparts in terms of OS thread usage and memory consumption. With our approach we focus only on threads, even if we collect memory consumption parameters.

The work of [14] contrasts the latency per request between Fiber, an alternate term for Java virtual threads, and traditional Java OS Threads. The paper notes that the existing Java thread model exhibits scalability issues in concurrent applications due to the excessive creation of OS threads during execution. HTTP servers were constructed using the standard library provided in the Java Development Kit (JDK). Each prototype server was assigned a wait time for incoming requests, with 75 requests per second sent to the HTTP server during each test run. Results indicate that when the sleep time per request was set to 1 ms, the latency disparity between the fiber implementation and the standard implementation was negligible. However, with a 10-millisecond wait time, there was a notable discrepancy in latency between OS threads and virtual threads. The OS thread exhibited a mean latency of 25 milliseconds, whereas the virtual thread implementation showed a mean latency of 4 milliseconds. With our experiment we somehow validate the conclusions of this work.

III. Experimental setup

In this section we will present our experimental setup based on: i) a set of 14 Java projects or analyzed systems; ii) a parser that converts the analyzed system's jar file into a weighted graph; iii) two algorithms HITS [3] and PageRank [4] applied on the weighted graph ranking the classes in their order of importance; iv) three implementation for each algorithm based on ST, PT and VT threading models; v) a set of four machines equipped with Java 21 (Open JDK and Oracle JDK) to run the algorithms on: "dsel" Debian Linux 4.9.0-19-amd64, 2 cores, 4 GB RAM; "aiw1" Debian Linux 5.15.131-1-pve, 4 cores, 32 GB RAM; "cs26" Windows 10, Intel64 Family 6 Model 58 Stepping 9 Genuine Intel, 4 cores, 8 GB RAM; "cs30" Windows 11, Intel64 Family 6 Model 140 Stepping 1 Genuine Intel, 8 cores, 16 GB RAM.

A. The set of analyzed systems

The list of the 14 analyzed Java object-oriented systems or projects is depicted in Table I.

TABLE I
ANALYZED SYSTEMS

System Link	# classes
Ant-1.10.jar http://ant.apache.org	524
ArgoUML-0.34.jar http://argouml.sourceforge.net	852
GWTPortlets-0.95.jar http://code.google.com/p/gwtportlets	222
Hibernate-core-5.2.12-final.jar http://hibernate.org/orm/releases/5.2	4450
jEdit-5.1.0.jar http://www.jedit.org	1266
jGap-3.6.3.jar https://sourceforge.net/projects/jgap	447
jHotDraw-6.0.jar https://sourceforge.net/projects/jhotdraw	398
jMeter-core-2.0.1.jar https://jmeter.apache.org	280
log4j-core-2.10.0.jar https://logging.apache.org/log4j/2.x/download.html	1019
Mars-3.06.jar http://mars-sim.sourceforge.net	1709
Maze-1.jar http://code.google.com/p/maze-solver	121
Neuroph-2.2.jar http://neuroph.sourceforge.net	275
tomcat-catalina-7.0.67.jar https://tomcat.apache.org	680
Wro4j-core-1.6.3.jar http://code.google.com/p/wro4j)	336

We will denote the analyzed systems using the first three letters from their names in lower case. E.g. Ant will be *ant*, ArgoUML will be *arg*, etc. The analyzed systems were chosen from various domains, each with a large range for the number of classes.

B. The HITS algorithm

The HITS (Hyperlink-Induced Topic Search) algorithm [3] is a link analysis algorithm used to rank web pages. It was developed by Jon Kleinberg in 1999. The algorithm assigns two scores to each web page: authority and hub scores. Authority Score is a measure of the quality and reliability of a web page's content. A page with a high authority score is considered to be a valuable source of information on a particular topic. Hub

Score is a measure of the page's connectivity or popularity, indicating how well it points to other authoritative pages on the same topic.

The algorithm iteratively computes these scores based on the links between pages. Pages with many incoming links from authoritative pages are considered to have high hub scores, while pages with many outgoing links to authoritative pages are considered to have high authority scores.

The HITS algorithm employs the following steps:

1) Initialization: Assign initial scores to all pages.
2) Iterative Update:
   - Update the authority score of each page by summing the hub scores of the pages linking to it.
   - Update the hub score of each page by summing the authority scores of the pages it links to.
   - Normalize the scores to prevent them from growing too large or small.
3) Repeat: Repeat the update process until the scores converge or reach a predefined threshold.
4) Ranking: Rank the pages based on their final authority and hub scores.

In both parallel implementations PT and VT we employ of number of threads equal with the number of nodes in the graph with no inter-thread interaction.

C. The PageRank algorithm

The PageRank algorithm [4] is a link analysis algorithm used by search engines to rank web pages in search results. It was developed by Larry Page and Sergey Brin, the founders of Google, while they were graduate students at Stanford University.

The algorithm assigns each web page a numerical score, called PageRank, which represents the importance of the page. The basic idea behind PageRank is that important pages are linked to by other important pages. Therefore, a page's PageRank score is determined by the number and quality of links pointing to it.

The PageRank algorithm employs the following steps:

1) Initialization: Assign an initial PageRank score to each page.
2) Iterative Calculation:
   - Calculate the PageRank score of each page based on the PageRank scores of the pages linking to it.
   - Update the PageRank scores iteratively until convergence is reached or a predefined number of iterations is performed.
3) Damping Factor: Introduce a damping factor (typically set to 0.85) to model the probability that a user will continue browsing by following links rather than jumping to a new page randomly.
4) Teleportation: Introduce a teleportation factor to account for the possibility of randomly jumping to any page, which helps to avoid issues with disconnected components in the web graph.
5) Ranking: Rank the pages based on their final PageRank scores.

In both parallel implementations PT and VT we employ of number of threads equal with the number of nodes in the graph with no inter-thread interaction.

IV. Experimental results

From each experiment on each machine we collect one CSV data file containing data about the runtime of the three threading models. The header of the data file contains the following fields: Project, No of Classes, Attribute, Time, Thread Type, OS Name, OS Architecture, OS Version, CPU Id, CPU Arch, CPU ArchW3264, Number of processors, Available processors/cores, Free memory bytes, Hostname, Set. Not all fields were filled depending on the operating system.

On machine "cs30" we obtained the following runtimes listed in Tables II and III.

TABLE II
Runtime for computing attribute AUTH-W

Prj	ST	ST%	PT	PT%	VT	VT%
ant	0.348	0.95%	1.301	3.22%	0.157	1.03%
arg	0.781	2.12%	2.024	5.01%	0.314	2.05%
gwt	0.03	0.08%	0.486	1.20%	0.015	0.10%
hib	29.082	79.07%	19.042	47.14%	12.271	80.20%
jed	1.369	3.72%	3.403	8.42%	0.502	3.28%
jga	0.33	0.90%	1.16	2.87%	0.094	0.61%
jho	0.337	0.92%	1.019	2.52%	0.094	0.61%
jme	0.11	0.30%	0.706	1.75%	0.031	0.20%
log	0.723	1.97%	2.719	6.73%	0.254	1.66%
mar	3.035	8.25%	4.91	12.15%	1.397	9.13%
maz	0	0.00%	0.298	0.74%	0	0.00%
neu	0.087	0.24%	0.691	1.71%	0.016	0.10%
tom	0.424	1.15%	1.76	4.36%	0.11	0.72%
wro	0.125	0.34%	0.879	2.18%	0.046	0.30%

TABLE III
Runtime for computing attribute PR-U2-W

Prj	ST	ST%	PT	PT%	VT	VT%
ant	0.691	0.66%	1.819	2.65%	0.22	0.60%
arg	1.554	1.48%	2.971	4.33%	0.662	1.82%
gwt	0.047	0.04%	0.674	0.98%	0.022	0.06%
hib	86.658	82.65%	39.381	57.37%	30.067	82.58%
jed	2.935	2.80%	4.503	6.56%	1.04	2.86%
jga	0.53	0.51%	1.478	2.15%	0.204	0.56%
jho	0.575	0.55%	1.34	1.95%	0.179	0.49%
jme	0.219	0.21%	0.917	1.34%	0.073	0.20%
log	1.617	1.54%	3.53	5.14%	0.535	1.47%
mar	8.895	8.48%	7.422	10.81%	3.028	8.32%
maz	0.01	0.01%	0.376	0.55%	0.008	0.02%
neu	0.119	0.11%	0.868	1.26%	0.052	0.14%
tom	0.783	0.75%	2.281	3.32%	0.238	0.65%
wro	0.221	0.21%	1.087	1.58%	0.083	0.23%

V. Discussion

In the followings, we will answer research question (RQ1), about comparing the runtime performances of ST and PT models, analyzing Figures 2 and 3.

The computation of AUTH-W attribute on all four machines using ST execution was faster than using PT execution, except for project "hib". The computation of PR-U2-W attribute on all four machines using ST execution was faster than using

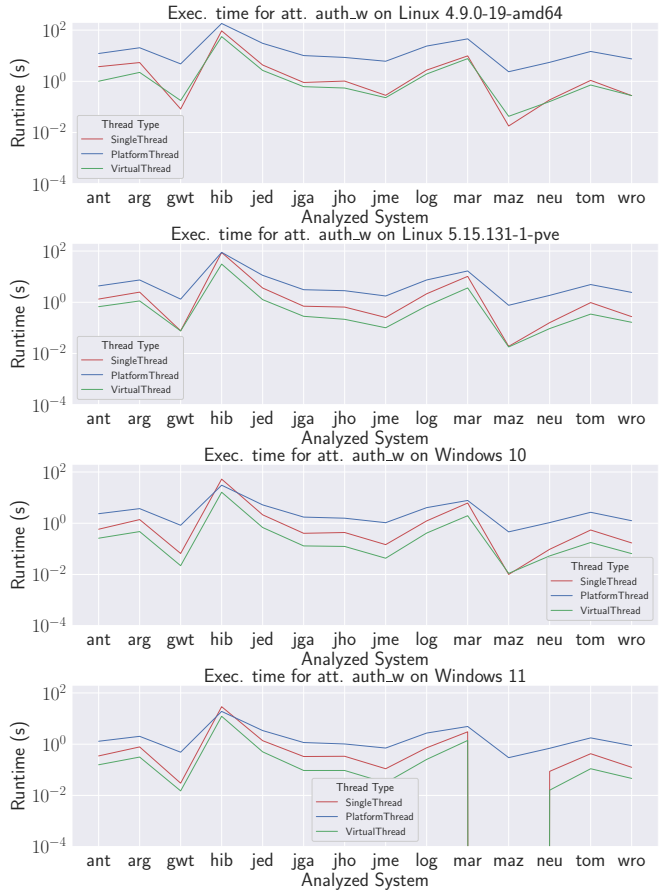

Fig. 2. ST vs. PT vs. VT Threading Model Runtimes for Attribute AUTH-W

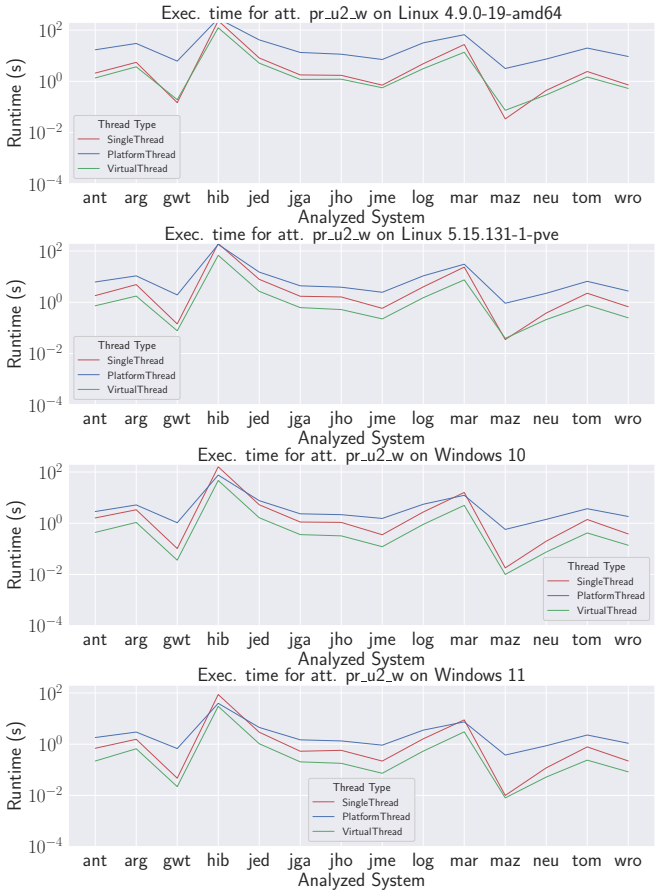

Fig. 3. ST vs. PT vs. VT Threading Model Runtimes for Attribute PR-U2-W

PT execution, except on two machines using Windows OS, both projects ”hib” and ”mar” are computed faster by the PT model. The two projects have the largest numbers of classes in the set, namely 4450 and respectively 1709. These projects form a cluster from this point of view. The scale chosen for the plots is logarithmic since the runtime differences of the algorithms running on different projects are large.

In the followings, we will answer research question (RQ2), about comparing the runtime performances of ST and VT models, analyzing Figures 2 and 3.

The computation of AUTH-W attribute performed faster using VT model than ST and PT models, except ”gwt” and ”maz” on the two Linux OS machines. On the Windows 10 OS machine the algorithm running on project ”maz” consumes a larger runtime for VT than ST.

The computation of the PR-U2-W attribute is performed faster by the VT threading model on all considered machines, except on the Linux OS machines for ”gwt” (”dsel”) and ”maz” (”dsel” and ”aiw1”). Project ”gwt” has 222 classes, while ”maz” has 121 and the runtime differences are not significant.

To answer research question (RQ3) we will compile the tables from the experimental measurements. The runtimes for computing attribute AUTH-W for the whole project set on all four machines are given in Table IV, expressed in seconds.

TABLE IV
RUNTIME FOR COMPUTING ATTRIBUTE AUTH-W

Thread Type / OS	Linux 4.9	Linux 5.15	Win 10	Win 11
SingleThread	123.281	110.529	66.041	36.781
PlatformThread	372.911	154.996	64.248	40.398
VirtualThread	73.958	39.659	20.627	15.301

The runtimes for computing attribute PR-U2-W for the whole project set on all four machines are given in Table V, expressed in seconds.

TABLE V
RUNTIME FOR COMPUTING ATTRIBUTE PR-U2-W

Thread Type / OS	Linux 4.9	Linux 5.15	Win 10	Win 11
SingleThread	287.992	245.898	192.799	104.854
PlatformThread	546.61	280.254	124.059	68.647
VirtualThread	155.77	84.945	57.203	36.411

From both Tables IV and V we notice that the ”cs30” machine, is the fastest employing the VT threading mechanism. This is a natural conclusion since the machine has a very powerful CPU with the following description: Intel64 Family 6 Model 140 Stepping 1 Genuine Intel.

We further notice that the runtimes to compute attribute AUTH-W are smaller than the runtimes to compute PR-U2-W. This could be explained by the number of iterations of the

two algorithms, the former having 20 iterations and the latter 50 iterations.

Research question (RQ4) is about identifying the projects consuming most of the runtime resources. The projects that consume most of the runtime resources when computing the attributes for the whole set can be identified in Tables II and III. Project "hib" consumes arround 80% while project "mar" arround 10%. The two projects "hib" and "mar" consume 90% of the total runtime of the whole set.

To answer (RQ5) about the significant runtime performance between the four machines there is no general conclusion to be drawn, but a particular one. The fastest "cs30" machine seems to be 3 to 8 times faster than the slowest "dsel" machine. The second fastest machine is "cs26" which seems to be 2 to 5 times faster than "dsel" machine. The "aiw1" machine is 1 to 2 times faster than "dsel". Probably, an optimal running scenario would be to run project "hib" on the fastest machine, while project "mars" on the second fastest machine and the rest of the projects on the remaining two machines.

To answer (RQ6) research question about clustering the projects using different threading models to minimize the overall runtime, we need to analyze the runtimes from Figures 2, 3 and Table VI.

TABLE VI
RUNTIME FOR BOTH ATTRIBUTES ON THE WHOLE SET ON "CS30"

Runtime	ST	PT	VT	min(ST,PT)	min(ST,PT,VT)
auth-w	36.78	40.39	15.30	26.74	15.30
pr-u2-w	78.06	92.75	32.45	57.98	32.45
Total	114.84	133.15	47.75	84.72	47.75
Rt. decr.		-15.94%	58.41%	26.22%	58.41%

One machine "cs30", using the ST and PT models we could minimize the amount of time for the set of projects by grouping "hib" and "mar" in cluster running on the PT model, and by grouping the rest of the projects into a second cluster running on the ST model. Thus, we obtain a runtime decrease of 26.22% against the ST model. Using the PT model we obtain a runtime increase of 15.94% against the ST model. This sub-performance could be explained by the time consumed on the thread stack operations. Using the VT model on all projects we obtain a runtime decrease of 58.41% against the ST model.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we benchmarked three threading models ST, PT and VT on four machines, two operated by Windows OS, and two operated by Linux OS. The experiments were completed in the context of two algorithms AUTH-W and PR-U2-W computing attributes useful in the automatic detection of key classes.

The conclusions are as follows. ST model is on 12 out of 14 projects better than PT. PT model performs better when computation is executed on more than 1,700 nodes, namely on two projects "hib" and "mar". The VT model outperforms both ST and PT. Clearly, the recent advent of VT model is a success.

As future work, we intend to reduce even more the runtime of the two algorithms using the threading mechanism offered by GPUs, through the CUDA API. We intend to implement the two algorithms AUTH-W and PR-U2-W using CUDA threads. Thus, we could explore larger solution spaces for the weight vectors generated by enumeration or heuristic algorithms.

REFERENCES

[1] Oracle Corporation. Project Loom. https://openjdk.java.net/projects/loom/. Accessed: [15.01.2024].

[2] Ciprian-Bogdan Chirila and Ioana Şora. The assessment of class relations importance using enumeration in a graph representation model for key classes detection based on pagerank. In *Proceedings of the International Conference on System Theory, Control and Computing (ICSTCC 2022)*, pages 1–6, Sinaia, Romania, October 2022.

[3] Jon M. Kleinberg. Hubs, authorities, and communities. *ACM Computing Surveys (CSUR)*, 31(4es):5, December 1999.

[4] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[5] Ioana Şora and Ciprian-Bogdan Chirila. Finding key classes in object-oriented software systems by techniques based on static analysis. *Information and Software Technology*, 116(UNSP 106176):1–20, December 2019.

[6] Weifeng Pan, Hua Ming, Dae-Kyoo Kim, and Zijiang Yang. Pride: Prioritizing documentation effort based on a pagerank-like algorithm and simple filtering rules. *IEEE Transactions on Software Engineering*, 49(3):1118–1151, 2023.

[7] Ciprian-Bogdan Chirila and Ioana Şora. Enumerating class relations weights to assess their importance in a graph representation model for detecting key classes using pagerank. In *Proceedings of the International Conference on System Theory, Control and Computing (ICSTCC 2023)*, pages 1–6, Timisoara, Romania, October 2023.

[8] Inc. Microsoft. Microsoft windows operating system. Computer Software, 1985. Version 1.0.

[9] Linus Torvalds. Linux operating system. Computer Software, 1991. Version 0.01.

[10] Jordi Guitart Fernandez. Performance improvement of multithreaded java applications execution on multiprocessor systems. https://upcommons.upc.edu/bitstream/handle/2117/93304/01Jgf01de01.pdf?sequence=1, 2005.

[11] Yo Han Joo and Carl Haneklint. Comparing virtual threads and reactive webflux in spring. a comparative performance analysis of concurrency solutions in spring, 2023.

[12] D. Beronić, P. Pufek, B. Mihaljević, and A. Radovan. On analyzing virtual threads – a structured concurrency model for scalable applications on the jvm. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1684–1689, 2021.

[13] D. Beronić, L. Modrić, B. Mihaljević, and A. Radovan. Comparison of structured concurrency constructs in java and kotlin – virtual threads and coroutines. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1466–1471, 2022.

[14] P. Pufek, D. Beronić, B. Mihaljević, and A. Radovan. Achieving efficient structured concurrency through lightweight fibers in Java Virtual Machine. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1752–1757, 2020.