IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 8 June 2023, accepted 21 June 2023, date of publication 10 July 2023, date of current version 24 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3293525

RESEARCH ARTICLE

ParallelC-Assist: Productivity Accelerator Suite Based on Dynamic Instrumentation

NACHIKETA CHATTERJEE [1], SRIJONI MAJUMDAR [2], (Student Member, IEEE), PARTHA PRATIM DAS [3,4], (Member, IEEE), AND AMLAN CHAKRABARTI [1], (Senior Member, IEEE)

[1]A. K. Choudhury School of Information Technology, University of Calcutta, Kolkata, West Bengal 700073, India
[2]School of Computing, University of Leeds, LS2 9JT Leeds, U.K.
[3]Department of Computer Science, Ashoka University, Sonipat, Haryana 131029, India
[4]Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India

Corresponding author: Nachiketa Chatterjee (nachiketa.chatterjee@gmail.com)

ABSTRACT Software developers often face challenges in terms of quality and productivity to match competitive costs. The software industry seeks options to minimize this cost during different phases of software development and maintenance with improved productivity. Software developers adopt different tools for different purposes, such as understanding program behavior, debugging memory issues, debugging concurrency issues, and testing. In this article we study different debugging tools mostly used for program design analysis, thread debugging, and resource management. Stand-alone tools do track static or dynamic control flow, thread activities, etc. But these do not specifically identify the thread work-breakdown-structure, global memory location management, thread-data interaction, etc. to allow good comprehension of the concurrency model of the program. Similarly for resource management, we observe that the `Valgrind` addresses a few required features but does not offer automatic garbage collection. Moreover, to address the outcomes of different tools, developers must compile and configure the application in different environments. This is very time-consuming, requires skills in different software paradigms, and is sometimes not supported by the tool itself. As a result, they cannot be used in an inter-operable manner to analyze by relating the different tool's outcomes. In this study, we conduct a detailed survey of the available tools and techniques and their limitations in identifying gaps. We address these gaps by implementing the tools for different phases of software development and maintenance. For example, a concurrency model detector based on thread behavior, resource debugger with features of automatic garbage collection, etc. can collectively inter-operate within our designed open-source tool framework ParallelC-Assist to address the common requests of the developers in one toolset. The tool is built upon open-source dynamic instrumentation tool PIN and supports a wide variety of IDEs and OS to detect various multi-threaded memory issues and provide additional features to inject concerns dynamically at run-time to extend it further according to the user's needs. We verify our tool with a wide variety of industry-standard benchmarks and compare its features with other similar tools.

INDEX TERMS Multi-threaded issues, memory issues, dynamic instrumentation.

I. INTRODUCTION

While the costs for Software Development Life Cycle (SDLC) have reduced considerably over the past three decades, the maintenance cost has gone up significantly and is amounting to more than 90% of the total SDLC cost [1], [2], [3] now. The National Institute of Standards and Technology, USA, estimates that 54.33% and 21.42% of the SDLC costs are related to the efforts of spent by developers to fix bugs and enhance code in the maintenance phase [4]. The costs are incurred owing to challenges in Program Comprehension (PC) of existing code bases, lack of adequate documentation, improper knowledge transfer from core teams, and incomplete and inadequate test strategies [1]. Naturally, this leads

The associate editor coordinating the review of this manuscript and approving it for publication was Hui Liu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

to lowering of the quality of software and the productivity of developers.

The manifestation, nature, and complexity of the challenges of comprehension vary widely on the type of programming languages. For example, the dangling pointer issue in C is automatically managed by the Java run-time. Hence, the nature of the support required by the developers also varies accordingly. C provides low-level access to memory and hardware, has cross-platform features, and is primarily used to build the firmware, operating systems, and so on [5]. For effectiveness, the codes written in C must certainly be multi-threaded in nature. A significant number of multi-threaded C codes have been developed in the last decade owing to the rise of cost-effective and energy-efficient multi-core technologies [6]. Due to this proliferation, many developers have had to deal with new correctness issues (like non-deterministic nature, concurrency bugs) and performance improvement techniques, without effective and simple multi-core programming tools, which in turn significantly added to the maintenance woes and overhead.

The major support areas for multi-threaded programming include analysis of concurrency-induced bugs, concurrency related design aspects, and performance improvement, in addition to the support required for single threaded applications such as debuggers and resource managers [7]. A survey conducted by Microsoft concluded that 66% of the developers find it difficult to deal with concurrency-induced bugs and issues and often need help to comprehend the concurrency models of an application to debug concurrency bugs. For example, to find the root cause of a deadlock and to fix the same, concurrency models related to the thread data (or resource) interaction, lock hierarchy, thread work breakdown structure, and starving threads need to be known.

Research has been conducted to help developers deal with the complexities of multi-threaded applications. Debuggers like Intel Debugger (IDB) [8] and Intel Inspector [9] provide advanced debugging features for threads (mostly data race) and memory errors. They provide APIs for integration with popular IDEs such as Eclipse but mostly through commercial product suites such as Intel Composer XE or Intel System Studio [10]. A number of approaches have been proposed for deadlock and data-race detection through the analysis of run-time events in [11], [12], and [13]. Apart from debugging, the analysis of thread activities and synchronized executions has been attempted in [14] and [15]. Researchers have employed static and dynamic weaving of code using aspects to understand the behaviour of the code for designing relevant test cases [16] or for extracting design elements [17]. Intel provides development suites like Intel Parallel Studio [18], and Intel System Studio [19] for debugging, testing, tracing, and monitoring applications. The design principles of existing tools and strategies used for multi-threaded debugging, resource management, design analysis, and dynamic aspect weaving, are detailed in Section II including the gaps identified in each area.

We observe that most existing approaches are standalone tools and address specific issues related to multi-threading. There is an absence of an integrated inter-operable framework to provide aid to analyze a multi-threaded application in its totality. Owing to the non-deterministic nature of multi-threaded applications, it is important to understand the design of the application to fix or to enhance them. In addition, the framework needs to be integrated with IDEs, to reduce the learning overhead of developers and facilitate easy diffusion [20]. An integrated framework would reduce the cross-transportation of data, overheads in learning new development environments, and multiple installations, and the like. Intel, and Microsoft, have tried to address this to a certain extent and have created development suites for bug-fixing, quality assurance, and testing strategies of multi-threaded applications with support for integration to standard IDEs and debuggers [21]. However, most of these frameworks do not focus on deducing the concurrency-related design aspects, are commercial, and cater to the needs of a certain language (mostly open MP), development environment, or compiler. They additionally lack the features for supporting customized tools and features, which might be required apart from the features supported in the suite.

In this article, we propose ParallelC-Assist tool set to analyze concurrency-related aspects of design based on thread-resource interaction [22]. We target to detect deadlock, data-race, and possible livelocks using GNU Debugger (GDB) augmented with new commands [12], to support interface for dynamic weaving to inject thread functions at run-time [23], and to automate garbage collection for C applications [24].

The tools related to concurrency models and dynamic weaving are of one type, and there are not many equivalent tools that analyze these aspects of the multi-threaded applications. Hence, integrating these tool sets with the concurrent bug detection and resource management tools makes the framework more effective. We have tested ParallelC-Assist using the pthread CDAC [25] benchmark. We ran all the tools individually, repeated the process through the integrated architecture, and obtained correct results for all the programs in the test suite. ParallelC-Assist can be extended to other OS, debuggers, or compilers based on the availability of suitable interconnection APIs. Hence, the major contributions of this study are as follows.

- Study of the various open source instrumentation tools, IDE's, and debuggers and their possible interactions
- An inter-operable framework of tools, integrated with common IDE's, to assist in developing and maintaining multi-threaded applications
- Unique combination of tools that deduce concurrency-related design aspects along with concurrency bugs

The remainder of this paper is organized as follows. Section II presents a literature survey. We discuss the architecture of ParallelC-Assist in Section III, and individual tools and some case studies are discussed in Section IV. Finally, we conclude with directions for future work in Section V.

TABLE 1. Thread and Resource Debugging Support in various tools in different IDEs or Product Suites shown in comparison with the features of parallel C-Assist. Further comparative information are given in Table 5.

Thread Debuggers

Thread Debugger	GNU (gdb) [28]	Intel (IDB) [8]	Microsoft Visual Studio [29]	Helgrind [27]	Intel Inspector [9]	Parallel C-Assist
Memory Related Breakpoints	✓	✓	✓	✓	✓	✓
Thread specific breakpoints	✓	✓	✓	×	✓	✓
Thread synch breakpoints	✓	✓	✓	×	✓	✓
Thread data sharing events	×	✓	✓	✓	✓	✓
Data race detection	×	×	×	✓	✓	✓
Deadlock detection	×	×	×	✓	✓	✓
Livelock detection	×	×	×	×	×	✓

Resource Debuggers

Resource Debugger	Intel Inspector [9]	C++ Validator [30]	Visual Studio Profiler [31]	Parasoft Insure++ [32]	Valgrind Memcheck [33]	Parallel C-Assist
Uninitialized memory	✓	×	✓	✓	✓	✓
Lost pointers	✓	×	✓	✓	✓	✓
Leaked Global Memory	✓	✓	✓	✓	✓	✓
Unreachable allocations	✓	✓	✓	✓	✓	✓
Automatic Garbage Collection	×	×	×	×	×	✓

Helgrind and Intel Inspector are not open source and come as a part of product suites
Open-source versions have limited features and not available for all compilers
ParallelC-Assist **can be easily integrated with standard IDE's**

✓- Feature Present × - Feature Absent

II. RELATED WORK

The tools in our ParallelC-Assist framework are designed for single as well as multi-threaded native C applications with focus on four major functionalities – debugging concurrency bugs, discovering design models, automating resource management, and providing a handle to weave code using aspects. Hence, we review the integrated product suites, IDE-supported features, and standalone tools and utilities that target to provide similarly functional support.

A. DEBUGGING

Standard debuggers provide support for breakpoints in memory-related errors, such as overflow and uninitialized access, in addition to data control and monitoring of related breakpoints. Further, some of the debuggers support breakpoints to trace thread data interactions and concurrency bugs (deadlock and datarace). We enumerate the standard debuggers provided as part of the **IDE's** or as **product suites** in Table 1.

Stand-alone tools have been designed to specifically support advanced debugging features, such as concurrency bugs. The authors of [13], [33], and [34] suggested approaches to detect data races by constructing happens-before graphs on runtime event traces. To detect the data-race, Christiaens et al. [35] employed different logical clocks over the collected run-time traces of send-receive events. In [36], Moiseev et al. detected data races in SystemC designs by static analysis of every program construct and event notifications.

Gaps: None of the stand-alone tools leverage and integrate the support from standard debuggers. We address this issue by extending the open-source `gdb` [27] debugger in [12] to detect data races and deadlocks for multi-threaded C applications using `PIN` [37]. Hence, we re-use the standard debugging features of the `gdb` [27] and add support to concurrency related debug features.

B. RESOURCE MANAGEMENT

Detecting errors such as uninitialized memory, dangling pointers, unreachable locations, and leaks in the stack and heap memory are common supports required in all phases of the SDLC. The resource management tools available as part of the **IDE** or as **product suites** are listed in Table 1 (Resource Debuggers).

Most of the tools discussed in Table 1 are commercial and require recompilation with specific libraries. `Valgrind` [38], however, is free and has multiple features to detect several memory management and threading bugs and is also used for program profiling.

As part of **stand-alone tools**, `Windbg` [39] provides complete memory statistics (address, length, and freed size) for the heap-allocated locations. `ccmalloc` [40] is a memory profiler that detects memory leaks and detects repeated deallocation of the same memory location. `LeakTracer` [41] extended `gdb` to print the allocated memory locations that have not been freed. `Memdebug` [42] tracks and logs (if desired) memory allocations and deallocations to infer memory leaks.

Gaps: Among the open-source and commercial resource management tools, `Valgrind` [38] provides most of the required features including detection of several memory management and threading bugs. It is also used for program profiling. However, it does not provide a comprehensive interface consisting of functionalities such as automatic garbage collection. We address the same in [24] based on `PIN` [37], wherein we extend the features provided by `Valgrind`.

C. DESIGN

Comprehending the design is essential for any code fixing / enhancement task. For example, while fixing a performance issue, developers must understand the control flow on the relevant code lines along with the design of the code.

Research has mostly focused around **standalone tools**, where approaches have been suggested for constructing control-flow graphs and detecting design patterns. The authors in [43], [44], and [45] constructed a set of cogent relationships between the components of a program and the elements extracted from the application domain ontology based on a static analysis of the source code. The design patterns (Gang of Four) were extracted from the source code for object-oriented languages, using source code parsing in [46] and [47].

In the case of a multi-threaded program, the design is defined additionally in terms of the aspects of concurrency ingrained in the application. These aspects of design are difficult to infer because of their non-deterministic nature

and cannot be directly understood from the extracted control flow. In [48], [49], [50], and [15], the sequence of the program execution is transformed, and the relevant sequences are extracted, such as event control flows, thread, routine, and class mapping using static analysis and dynamic profiling. In [51], the authors estimated inactive threads to comprehend the effectiveness of parallelism in programs using dynamic profiling. The runtime patterns for thread behavior in the case of shared data locations were deduced by inspecting synchronized executions in [14] and [15].

Gaps: The tools proposed so far track static and dynamic control flow graphs, thread activities, execution sequences but do not aid to comprehend the concurrency-related design issues in totality. For example, understanding thread work-breakdown-structure, global memory location management, thread-data interaction, and thread scheduling is as important as understanding the design of a multi-threaded application. We address the same in [22], where we build a concurrency model detector based on thread behavior.

D. CODE WEAVING AND INSPECTION

Approaches have been explored for building an *Aspect-Oriented Programming* (AOP) framework using Java to help weave code, enhance or observe program behavior, and write relevant test cases [16]. AspectC++, an extension of AOP for C++, was created [52] based on AspectJ [53] of Java, to enable the static weaving of code. AspectC++ has been used in multiple scenarios; however, static weaving requires recompilation after every code change. These methods do not implement weaving-on-the-fly (without recompilation), that is, aspect weaving at runtime. We propose dynamic aspect weaving for C programs in [23], wherein we use dynamic instrumentation framework to attach, detach, and modify concerns during the execution of the program without modifying the program. Using the observations from code weaving, we can design effective test cases. We also observed that research related to code weaving has mostly focused on **standalone tools**.

E. INTEGRATED DEVELOPMENT ENVIRONMENT, PRODUCT SUITES

The `Eclipse CDT` [54] is a commonly used IDE that supports multiple features, such as call graphs, code highlighter, code generation, and debugging facilities, to help developers write correct and efficient code. IBM extended Eclipse to `Hyades` [55] with an integrated test and verification module. Microsoft has developed `PREfix` [21], an integrated framework that helps to detect logic and coding errors based on pattern matching with a pool of common errors. `PREfast` [21], another integrated tool from Microsoft, detects discrepancies in the coding conventions used. In [56] Microsoft presented `SLAM` for model checking, along with detecting common errors such as buffer overruns. SUN extends `Netbeans` to develop jackpot source code metrics to examine codes and detect structural issues [21]. Intel provides development suites like `Intel Parallel Studio XE` [57], and `Intel System Studio` [58], with support for tracing programs, analyzing execution sequences, detecting memory and thread errors, etc. The Intel frameworks also provide extension APIs for integration into standard IDEs, such as `Eclipse CDT`, `Visual Studio`, and debuggers, such as `gdb`.

The integrated frameworks, however, are mostly commercial and targeted to managed languages, such as Java and Python. Additionally, they focus on bug checking and program tracing and not on other crucial support, such as extraction of the design. Similarly, standalone tools focus only on specific aspects, incur a huge installation and learning overhead, and do not allow the facilities from other tools to be used in an inter-operable manner. For example, while debugging an application for a memory error, the developer may need to know the thread work-breakdown-structure or the likely threads that start in execution for a particular input. In another scenario, the developer may want to debug the program after dynamically weaving some code at runtime.

We address these challenges in ParallelC-Assist, where we integrate the support for debugging, design extraction, and code inspection into a singular framework and extend the same with a standard IDE such as `Eclipse CDT`. In the next section, we discuss the various open-source frameworks explored to conclude on a set for developing ParallelC-Assist.

F. SURVEY OF OPEN SOURCE FRAMEWORKS

As several design aspects of multi-threaded applications manifest only at runtime, we focus on open-source dynamic instrumentation tools to trace and extract runtime events. Further, there should be support for plugins for the integration of the instrumentation framework with a common development environment and debuggers. These plugins / interconnections must be re-targeted in order to develop an integrated framework. We review the tools and their support for extensibility into other frameworks in Table 2.

We see that `PIN` [37], [59] and `Valgrind` [38], [62] are both well-accepted and widely used dynamic instrumentation frameworks. However, `PIN` is lightweight, is 3.3x times faster [59] than `Valgrind`, has support for integration with multiple debuggers and IDEs, and is available for several operating systems. As our focus is on creating a framework that can benefit parallel developers working with multiple OSs, compilers, IDEs, and debuggers, the `PIN` is well-suited for our requirements. We use the `PIN` to build our analysis tools and re-target its remote extensions to `Eclipse CDT`, `gdb`, and `LLDB` debuggers to develop an integrated architecture. We explain the process of re-targeting and analysis tools in Sections III and IV, respectively.

III. ARCHITECTURE

Our aim is to develop an integrated architecture to support all features from a single screen without the need for cross-transportation of information and manual linking.

TABLE 2. Availability of plugins/interconnections for dynamic instrumentation tools for C, C++. We use PIN as shown in Figure 1.

Instrumentation Frameworks	OS	IDE	Debugger	Compiler	Thread Libraries
`Pintool` (Intel) [38], [60]	Windows, Linux, Android, MacOS	`Eclipse CDT`, `Code Blocks`, `Visual Studio`	`gdb`, `LLDB`	icc, gcc	pthread, boost, intel tbb
`QBDI` [61]	Windows, Linux, Android, MacOS	`Ninja`, `Visual Studio`	`LLDB`	gcc	pthread
`DynamoRio` [62]	Windows, Linux	`Eclipse CDT`	`gdb`	gcc	pthread
`Valgrind` [39], [63]	Linux	`Eclipse CDT`	`gdb`	gcc	pthread, boost, intel tbb

We first discuss a common scenario in the workplace as learnt from multiple developers of companies working in Electronic Design Automation extensively using C.

A. CASE STUDY OF INDUSTRY PRACTICES

We enumerate and analyze the problems faced by developer *Sandra* (name changed) while fixing a *deadlock* bug or analyzing function exit and entry points in a multi-threaded application, as shown in Table 3. Sandra starts the analysis to detect potential deadlocks in the application and tries to simulate various other run-time facts during program execution. She is challenged with a suitable toolset for the OS, or IDE she is working with. The case studies help in designing our architecture according to the developers' needs.

B. ARCHITECTURE

We design an architecture to fit all the individual tools together and integrate them to extend the assistance from a single window so that developers may benefit from using the tools simultaneously, such as detecting memory issues in addition to parallel debugging.

We present an overview of the architecture of the plugin toolkit in Figure 1. Our toolkit is based on concept extractors that are built using a `PIN` framework coupled with an inference engine. We integrate the toolkit with the `Eclipse IDE` using its plugin interface. Figure 2 shows the architecture of the `Eclipse IDE` [64]. The IDE has a workbench containing editors, consoles, and sits on a Java run-time for its utilities. Project management is also conducted using `WorkSpace` [64]. The plugin interface is available to add separate menus to the editors to support the additional features we are integrating.

To model the components of the architecture and develop an analytical framework, we define the following domains:

- $\mathcal{IG}$: Domain of APIs for Image Instrumentation
- $\mathcal{RT}$: Domain of APIs for run-time Instrumentation
- $\mathcal{IN}$: Domain of APIs for Instruction Instrumentation
- $\mathcal{D}$: Domain of various data structures in C Programs
- $\mathcal{OP}$: Domain of various data structures related to the operations in C

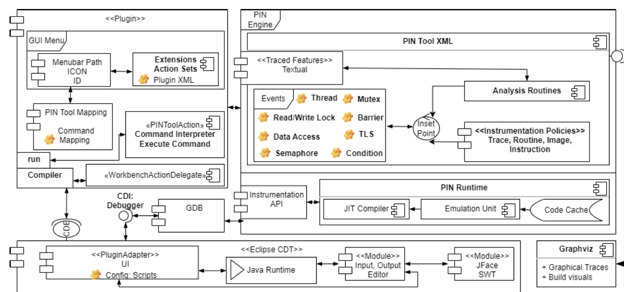

FIGURE 1. Integrated Architecture of ParallelC-Assist framework showing interactions between the components of IDE [Eclipse], Plug-ins, `PIN` Dynamic Instrumentation tools, and its run-time instance. Component-wise details are given in Figures 2, 3, and 4.

- $\mathcal{AS}$: Domain of analyzed aspects

An instrumentation process registers callbacks to either the image, routine, or instruction APIs and passes the pointer to the current object (image, routine, etc). It is defined as:

$\mathcal{I} = (AP,\ O)$, where AP is the API for image, routine, and instruction and O is the pointer to the current object.

The domain for AP is $\mathcal{D}_{ap} = \mathcal{IG} \cup \mathcal{RT} \cup \mathcal{IN}$

The inference process can thus be modeled as:

$\mathcal{A}_n : \mathcal{D}_{ap} \times \mathcal{OP} \times \mathcal{D} \times PIN \rightarrow \mathcal{AS}$ where PIN symbolizes the just-in-time compiler.

The equation focuses on the domain dependencies of the analysis process and generalizes the domains that must be considered for any inferences using the proposed framework.

In the next few sections, we explain the general working of the following sub-components:

C. THE PLUGIN INTERFACE

Our tool use the plugin interface (Figure 3) to act upon the executable from the editor of Java `Workbench` [64] and obtain the results. Hence, we partition the architecture into the following components to handle the various sub-tasks.

1) *Creation of graphical menu*: This is created as a new extension in the `Plugin.xml` file of the JDK package. Each new menu is an action set for extension. The menu is characterized using the functions of the label and icon class of the Java plugin.
2) *Link of menu to Pintools*: Each menu has a script at its back-end, which fires a Pintool that works on the current executable from the editor. Each menu first calls the required function from the action set class, which then fires the script.
3) *The plugin of Pintools with executable*: The Pintool Action class [64] is called when the script is fired for each menu. This class takes the source code as input and the required arguments from the editors, and compiles the program into an executable. It then traverses the pin executable engine through a script and plugs the relevant Pintool (analysis tool) with the executable.

D. `PIN` FRAMEWORK

We describe the `PIN` framework as re-targeted and customized for ParallelC-Assist in Figure 4. We use `PIN` to

TABLE 3. **Case studies for *detecting potential deadlocks* and *analysing function entry and exit points*. These studies were conducted with C developers from companies working in electornic design automation.**

Task at Hand	Possible Solutions Explored	Decision
Case Study 1: Developer *Sandra* wants to check for potential deadlocks		
Initial Exploration		
Sandra tries in a multi-threaded (pthread) C application, works with gcc compiler [64], Eclipse CDT 8.4 [65] & Ubuntu 16.04 OS	looks for standard toolset provided by Intel or Eclipse	finds a 30 days free trial pack of Intel Inspector [9], with extensions to Eclipse CDT
Installing and Using a new tool		
Eclipse CDT 8.4 is compatible with the automatic integration of Inspector XE into the Eclipse IDE [65] during installation. The version downloaded for Intel Inspector does not support gcc compiler	checks the supported compilers, icc [64] is found suitable	*Sandra* compiles the application with icc [64] and repeats the process
Intel Inspector finds two deadlocks but could not find potential deadlocks which might have not occurred in that run. *Sandra* wants to first fix the detected deadlocks		
Sandra looks for tracer tools to dump the run-time events in every run, wants to understand lock sequences, wants to know how long a thread is waiting on a synchronization resource	checks if Intel Inspector XE supports tracing, and also other tools	Intel Inspector XE does not support tracing but is a component tool of the Intel Parallel Studio, which has a tracer tool Vtune profiler [66]
Sandra explores Vtune profiler	Vtune works only for OpenMP-MPI and Intel-TBB frameworks	discards Vtune profiler [66]
Sandra continues to look for tracers	looks for other concurrency visualizers like Microsoft concurrency visualiser [14] for thread data executions. The visualizer mostly addresses thread contention, cross-core thread migration, and synchronization delays and comes integrated with Visual Studio	Shifts to a new IDE
Shifting to a new IDE		
Sandra uses the visualiser tool and find some feature useful. However, she now needs to find deadlock detection tools compatible with Visual Studio [31]	Explores various open source tools, find a relevant (finds potential deadlocks also) tool software verify, which can be integrated with Visual Studio	However, software verify [67] is priced at $499. Another trial version is available, with lots of forms to fill up, with a detailed reason for its use. Discards `software verify [67]`
Sandra tries to look for the free version of Intel Inspector XE, which can be integrated with Visual Studio [31]	find a patch version after a lot of manual searches	installs and integrates with difficulties
Even with the new set-up, a lot of information is required but missing to analyse and fix deadlocks – graphical traces, concurrency models- thread work breakdown structure, thread starvation. The features in Concurrency Visualiser [31] are mostly related to optimising code by checking the thread to core mapping and related statistics. This will not completely help to fix the deadlocks in a short span of time		
Sandra tries to look for more standalone tools	She checks research papers and the link to source code, but in most cases, the links are broken, or the repository is not up to date	Manually analyses the application and long with the output from Concurrency Visualiser and Intel Inspector XE, tries to fix the deadlocks
Case Study 2: Sandra wants to analyse the function entry and exit points		
Sandra tries to look for more tools that may work collectively on the same execution	No automated support	Manually analyses the output traces of the application as extracted from Helgrind [27] and Intel Inspector XE, to mark and log the entry/exit point of the functions
During execution Sandra realized that she needs to simulate an exception that was intermittently happening in a function leading to a critical issue		
Sandra searches for tools that may inject exceptions in run-time along-with existing execution	Could not find run-time injection tool with trace support	Manually added additional code to throw an exception in the source, re-compile, and re-execute

extract the primitives of the program and further analyze the same for higher-level features. Using the APIs of this framework, we created Pintools to extract and analyze the run-time traces of an application. The `PIN` [37] API's sit on a `PIN` run-time with the support of just-in-time compiler (JIT), an emulator, and a dispatcher.

1) PINTOOLS

Every Pintool has two major components:

1) Instrumentation routines: This specifies the uniformity at which run-time traces are collected. The available granularities include images, traces, routines, and instructions.
2) Analysis Routines: For every instrumentation granularity, there can be multiple analysis tools that store the traces collected in a data structure and analyze the same based on algorithms designed by us.

The `PIN` engine, along with the Pintools, interacts with the plugin interface and serves as input to the inference engine. The high-level features analyzed from the primitives in the analysis routines serve as the input to the inference engine.

E. INFERENCE ENGINE

The inference engine (Figure 1) contains a suite of machine learning classifiers that work on the features generated from Pintools and learn and predict a model. The engine also contains other algorithms based on a set of rules to detect execution behavior, which works directly on the primitives extracted from Pintools.

F. USER INTERFACE

Keeping tool usability in consideration, we leverage the `Eclipse IDE` and develop plug-ins for our tool assembly. In the plug-ins, the individual tools are designed as

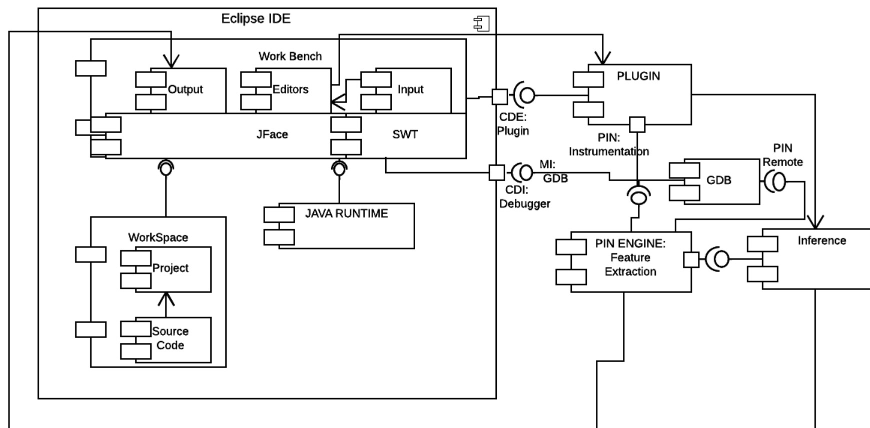

FIGURE 2. Architecture and interface diagram of Eclipse IDE platform including Workbench and UI Toolkits containing editors, consoles, and sits on a Java run-time for its utilities. The CDE plugin interface is available to add separate menus to the editors to support the additional features by interacting with `PIN` run-time interface and visualize acquired knowledge about a program from GDB callbacks thrugh CDI Debugger. This is a component from Figure 1.

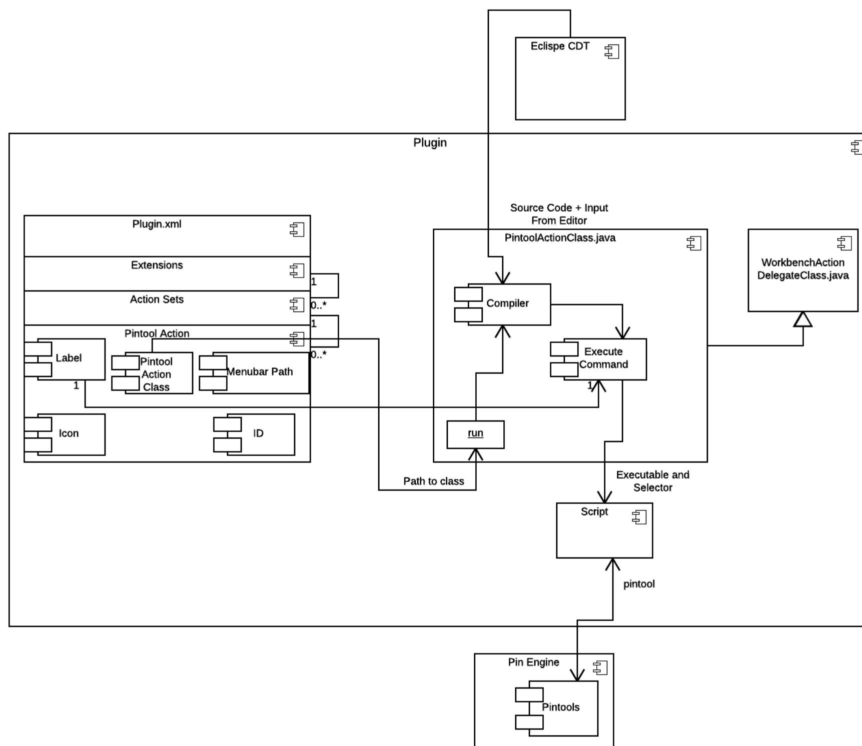

FIGURE 3. Flow diagram of Plugin Interface to inject tools on executable from Java Workbench editor and UI toolkit of Eclipse IDE. Each menu in the UI toolkit will have an associated script to fire a tool in `PIN` that works on the current executable from the editor. This is a component from Figure 1.

user menus, as shown in Figure 5. Users can select any combination of tools according to their needs by clicking on either icon or menu options. If the user clicks on GC, PGDB, and then on the tool assembly, the GC and PGDB execute in parallel to detect the data race or deadlock along with the detection of memory issues with optional garbage collection. Based on the selection of plug-ins, the code is selectively equipped with the required parts of unit tools. The instrumentation gathers run-time information of the program execution. The run-time console logs generated from the code and tool are combined into an Eclipse console. An inference drawn by the tool assembly is also prioritized, and suitable assistance, either printed on the console or breakpoint, may be invoked.

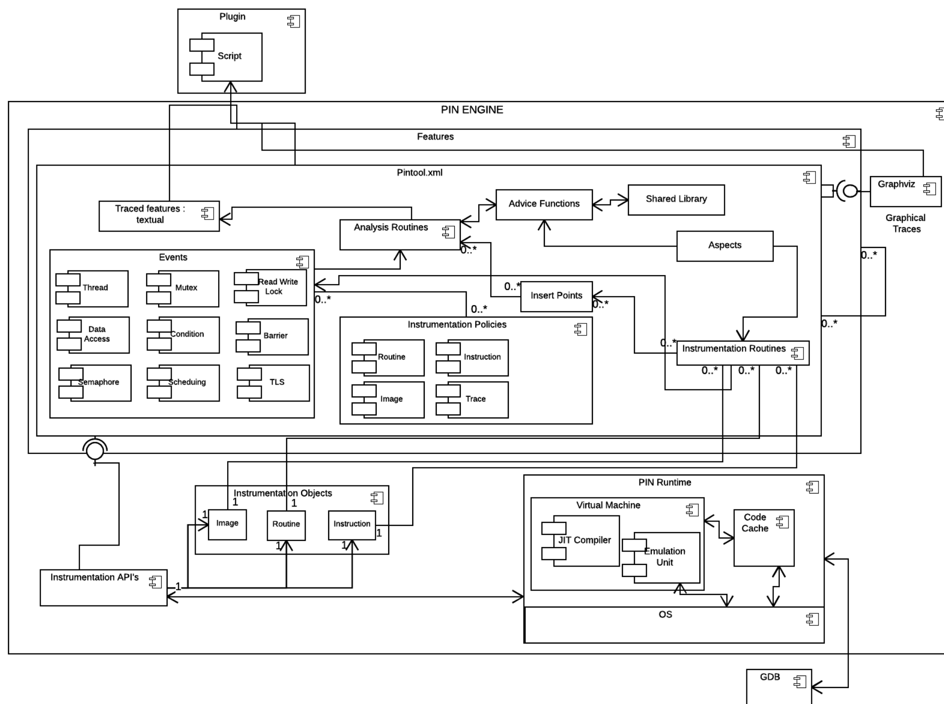

FIGURE 4. PIN **Infrastructure diagram of instrumentation components with Instrumentation API, Features and run-time interfaces.** PIN **Instruments the target application based on the Instrumentation policy to extract the run-time features and are analyzed using the analysis routine. At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instrumentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application [37]. This is a component from Figure 1.**

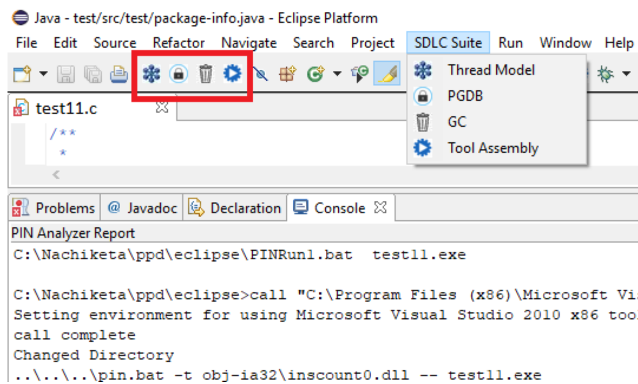

FIGURE 5. SDLC Suite menu and icons injected in Eclipse IDE as an User Inference of Tool Assembly to dynamically add or remove different debug tools or aspects as and when required.

An integrated view of the architecture is shown in Figure 1. The PIN engine contains various callbacks to functionalities, such as extracting images, routine, thread, and instruction level aspects, which are then deployed to the plugin architecture and work on the current executable of the IDE. The output from the Pintool is then integrated with a visualization software to display the run-time event traces. Furthermore, an enhanced debugger with support for additional concurrency bugs is integrated with the IDE to provide end-to-end support.

IV. THE TOOL SET

We discuss the various tool supports as provided in ParallelC-Assist: Debug assistance tool (Section IV-A), Design tool (Section IV-B), Memory tool (Section IV-C), and Aspect injection strategy (Section IV-D).

A. DEBUG ASSIST [12]

In the PGDB [12] tool, we designed features to detect and solve issues such as deadlock and data race with breakpoints to the source. We augment PGDB [12] with LLDB [67] and GNU debugger so that developers can leverage the facilities of PGDB [12] within their existing debuggers.

1) IMPLEMENTATION

Our approach is to identify whether there are memory references shared among multiple threads. We instrument RecordLockBefore() to monitor the locality of accesses with or without locks by a thread and maintain an hash table MemTracker, where the key is a memory reference and the value contains identification of executing thread and the type of access (READ/WRITE). We have designed the instrumentation routines RecordMemRead() and RecordMemWrite()

before load and store instructions, respectively, including the thread id to trace memory accesses from concurrent execution. First-time READ accesses to any memory reference are captured in MemTracker by analysis routine RecordMemRead(). For subsequent accesses to this captured memory reference that already exists in MemTracker, the following situations may occur [12].

- **Existing READ access**: This is a safe access. Thus, there is no data race in the case of READ-after-READ.
- **Existing WRITE Access**: For the same thread ID, it is a safe access case for READ-after-WRITE. For different thread IDs, the memory reference is marked as a shared-exclusive memory.

Similarly write accesses to memory being analyzed by RecordMemWrite, executes before *Store* instruction. First-time memory WRITE access is also captured by the MemTracker, including the thread ID. Again, subsequent accesses to the captured memory reference already exist in MemTracker and the possibilities are as follows:

- **Existing READ or WRITE Access**: Unsafe access in both cases, WRITE-after-WRITE or WRITE-after-READ. If the threads involved are different, the memory reference is marked as shared-exclusive memory.

A Boolean variable is introduced here for each thread to detect the datarace. When a thread, say *T1*, enters the critical section, RecordLockAfter() is called, and sets a flag for thread *T1*. While leaving the critical section for thread *T1*, we reset the flag using RecordUnlockAfter(). We also instrument the barrier along with thread ids. The inference block is used to analyze the memory read-write sequences of each thread. Therefore, access to a shared-exclusive memory reference is identified as unsafe, where the flag is set to false, and safe otherwise. Once a memory reference is identified as shared-exclusive and unsafe access exists, there is a potential for datarace and datarace breakpoints to be invoked.

Similarly, an algorithm continues to construct an RAG[1] by identifying the waiting and acquired edges as follows:

- RecordLockBefore() adds an edge to the RAG denoising thread T is waiting for resource (mutex) R, when another thread already holds the lock on mutex R.
- RecordLockAfter() adds an acquired edge to the RAG when t acquires a lock on the mutex r. The waiting edge was removed if an acquired edge was added.
- RecordUnLockAfter() removes the acquired edge from the RAG when thread t releases mutex r.

The inference block here continues to analyze the RAG, and once it detects a cycle, then announces a deadlock, and the breakpoint is invoked.

2) VERIFICATION

We prepared a test suite to determine the correctness of our tool for the detection of dataraces and deadlocks. The tool successfully detects all potential dataraces and deadlock conditions. Detailed test evidence is provided in the PGDB [12] and extended debugger [68] to verify the PGDB on the collected set of benchmarks and prove its behavior and efficiency.

[1] Resource Allocation Graph.

B. DESIGN TOOL

Here, we outline the design to capture the execution sequences from various PIN events.

1) IMPLEMENTATION

Here the run-time information is stored and grouped logically into maps, and then into profilers that produce output for a code related to a specific classified set of problems.

- We capture every important routine executed as part of a code as an event *s* with various parameters.
- We need to provide a logical ordering of events that might be useful for the debugging.
- Only those events that are relevant to debugging should be tracked, so we instrument only important routines from the total routine trace.
- To decide on the routines, we consider those related to thread creation, communication, data variables used in message passing, thread exit sequence, synchronization functions, and signaling functions.
- For every routine, we capture all the relevant information in the form of parameters such as the thread id and the data variable involved before and after the execution of the event, along with logical order.
- Storing the information extracted from routines related to various run-time occurrences, such as global variable access sequence, creation sequence of threads, communication statistics among threads, wait time of a thread to acquire a mutex, etc. by logically grouping them into distinct maps (data structure). The first level is a detailed map.
- Logically converting the detailed maps to summarised maps to extract distinct run time information.
- We provide the output to the user in textual and graphical forms.

2) VERIFICATION

For normal (error-free) execution, models detected from features extracted by our analyzer for a classified set of problems from the test suites are validated by models detected by manual analysis of the test suites. We also compare the results with existing analyzers such as Valgrinds, Purify, and other profilers, and achieve 91% accuracy [22].

C. MEMORY TOOL [69]

We designed a resource management tool for native languages that may be invoked as and when developers want to debug memory issues or manage resources. We call it **GC Pintool**, as it identifies the memory issues during the execution. Moreover, it offers optional garbage collection features in the native application:

1) IMPLEMENTATION

The GC tool is one of our unit tools used for tool assembly. The strategy of this tool was designed using two different components: instrumentation strategy and inference. First, we construct the instrumentation algorithm as follows:

- `main()` and user functions are instrumented to track the entry-exit points.
- On call of memory allocation, experiencing a local pointer, the reference is logged in the data structure with local scope. For a global pointer, the reference scope changed to global. For another assignment, scope is modified accordingly.
- In the deallocation of memory, the entry of the reference is removed from the data structure. If no such entry is found in the data structure, then a double-free error must be declared.
- The exit of each reference invokes an inference routine.

In the inference routine, the data structure is analyzed to identify the remaining entries associated with recently exited scope. Those entries are marked as memory leaks. These memories may be optionally freed up.

2) VERIFICATION

The efficacy of the **GC Pintool** was verified using different benchmark C programs, and the approach was proven to be correct and precise. From the literature survey, we found that `Valgrind` is a winner in comparison to other memory tools; therefore, using this tool, we offer memory error detection features, such as memory leak, memory corruption, double frees, and uninitialized pointers, along with breakpoints similar to `Valgrind`. `Valgrind` experiences a 10–50 times slowdown,[2] whereas **GC Pintool** always performed correctly, we find that it runs about 35% faster compared to `Valgrind`. Our tool also has optional automatic GC features. In this tool, we use a map as a core data structure to hold the scope and memory addresses belonging to a particular scope. We then devised an algorithm to instrument the memory allocator and deallocator functions to capture the allocation / deallocation events. Memory read and write events are captured by the incoming memory read and write instructions. We also investigated the function call and return events of the program scope. Upon entering a new scope, the new key will be defined in the map as the current scope, making the earlier scope a parent, and for each successful allocator execution, the allocated memory address will belong to that current scope in the map. For each deallocation, the memory reference is removed from the current scope. The inference block analyzes the allocated memory of a particular scope at the end of each scope. If any allocated memory is found to be dereferenced due to the end of scope, the memory leak is reported, and optionally, a breakpoint may be invoked or the detected leak may be garbage collected, that is, freed up by this tool. Similarly, the inference block also analyzes the reference of memory read or write; if the reference is outside our recorded memory references allocated during the execution, it declares this as memory corruption. In this case, the user could optionally invoke a breakpoint.

[2] "2.1. What `Valgrind` does with your program" in http://valgrind.org/docs/manual/manual-core.html

D. AOP - ASPECT-ORIENTED DEBUGGING [23]

In this suite, we added the framework of dynamic aspect weaving to extend the tool according to user needs. Here, the framework is flexible for the vanilla deployment of dynamic aspects in just-in-time.

1) IMPLEMENTATION

This framework works with the components below:

- The configuration XML is available for the user to define the function / event of the executable, that is, to be observed–advice names–what to observe, and the location (after/before)–where to observe.
- The analysis code library is compiled and maintained in the vanilla scope and is loaded on the fly in execution time at the placeholder into the desired function/event.
- Our `PIN` tool reads the configuration from XML and injects advice at desired locations in the code under execution by using the dynamic instrumentation technique.

2) VERIFICATION

We used a testbed to verify the injection of advice at various desired points. We successfully verified the location-before and location-after instrumentation for global functions, static functions, function pointers, and references in the C program. However, we identified the limitations of the tool in the case of macros. As we are performing run-time binary instrumentation, we do not have any control over the macro during the run-time. On the other hand, in C++, we successfully injected advice in Constructor, Overloaded Constructor, Copy Constructor, Destructor, Overloaded Operator, etc. We also validated the injection of aspects in the Friend Function, Member Function, Overloaded Member Function, Virtual Member Function, and Overridden Member Functions and identified some limitations for the in-line function. We used our tool for system and user-defined functions in exception scenarios for different hierarchies. The same tool has been used to inject bits of advice for different thread events, such as create, join, lock, and unlock. In all of the above scenarios, we successfully injected the aspect before or after the events occurred.

E. TESTING THE INTEGRATED ARCHITECTURE

We tested the integrated architecture of ParallelC-Assist on the pthread CDAC [25] benchmark for correctness and robustness. The benchmark set of CDAC contains a varied set of programs that replicates various concurrency-related issues and bugs, numerical computations using threads in parallel, input / output, and other resource management using

TABLE 4. Concurrency characteristics of the testsuite of PARALLELC-ASSIST from CDAC Pthread Benchmarks [25]. We achieved an overall precision and recall score of 98.13% and 98.56%, respectively on these.

File Name	PGDB [23]		D-Cube [22]			GC [25]
	Data-Race	Deadlock	Thread Concurrency	Global Variable	Starvation	Leak
Basic Pthread API calls						
Eventual Starvation and Livelock considered for a certain run and input						
Datarace and Deadlock are also input and run specific						
Pthread Join	No	No	Boss Worker	Only Locals	No	Thread local
Stack Management	addr 0x78C741; func `dowork`	No	Peer to Peer	Unsync, Low Locality	No	No
Mutex Operation	No	No	Boss Worker	Sync, High Locality	No	No
Condition Waiting	No	No	Pipeline	Sync, High Locality	No	No
Disjoint Array Access	No	No	Boss Worker	Sync, High Locality	1 thread	Leak
Numerical Computations (Dense Matrix Computation)						
Numerical Integration	No	Yes	Boss Worker	Sync, High Locality	No	No [fn]
Vector Multiplication block striped partitioning	No	No	Boss Worker	Sync, High Locality	No	Function & Thread Local
Infinity norm - Row-wise Partitioning	No	No	Peer to Peer	Sync, High Locality	Yes	Function & Thread Local
Infinity norm - Column-wise Partitioning	No	No	Peer to Peer	Sync, High Locality	Yes	Function & Thread Local
linear equations - Parallel Jacobi Method	addr 0x12A89C; 0x 12A90B, func `Jacobi`	No	Boss Worker	Unsync, High Locality	No	Function & Thread Local
Non-Numerical Computations & I/O – Sorting, Searching, Producer-Consumer, using thread APIs						
Minimum in unsorted array	No	No	Peer to Peer	Partial Sync, Low Locality	No	No
Producer-Consumer work queues	No	No	Boss Worker	Sync, Low Locality	No	Thread Local
k matches in the list	No	No	Peer to Peer	Partial Sync, Low Locality	No	No
data race condition	addr 6234231; func: `thread_mutex`	No	Peer to Peer	Partial Synch, High Locality	No	No
Loop-carried dependence to Loop-independent dependence	No	No	Boss Worker	Unsync, Low Locality	No	Function Local
Read-Write API library calls						
Read-Write locks	No	No	Boss Worker	Sync, High Locality	No	No
array minimum - Read-Write locks	No	yes [FP]	Peer to Peer	Sync, High Locality	No	No
array minimum - Read-Write & mutex locks	No	No	Boss Worker	Sync, High Locality	1 thread	No
Illustration of Producer/ Consumer problems using pthreads for large queues						
producer/consumer; Indexed-access	No	No	Boss Worker	Sync, High Locality	3 threads	Thread Local
producer/consumer; Condition waiting	No	Yes [FP]	Peer to Peer	Sync, Low Locality	No	Thread Local
producer/consumer; Mutex objects	No	Yes	Boss Worker	Sync, High Locality	No	Thread Local
producer/consumer; Condition-variable	No	No	Boss Worker	Sync, High Locality	No	Thread Local

threads and the like, and helps us to test ParallelC-Assist for utility. The results are presented in Table 4. The results were validated against the benchmark documentation available with the dataset. We achieved an overall precision and recall score of 98.13% and 98.56%, respectively (false positives and false negatives are marked as FP and FN in Table 4). As ParallelC-Assist also detects a potential deadlock, it might happen that the deadlocks do not occur in a particular run and hence the false positives. We get a false

TABLE 5. Comparison with existing frameworks as outlined in Table 1.

Tools	Visual Studio [72]	NetBeans [73]	Eclipse CDT [65]	Code:: Blocks [74]	Intel Debugger [75]	GDB [28]
Rational Rose [76]	✓	×	×	×	✓	×
MS Concurrency Visualizer [14]	✓	×	×	×	✓	×
Bottle Graph [77]	×	×	×	×	×	×
Visual Leak Detector [78]	✓	×	×	×	×	×
GNU Checker [79]	×	✓	×	×	×	✓
AspectC++ [53]	×	×	×	×	×	✓
ThreadSanitizer(TSAN) [80]	×	✓	×	×	✓	✓
Helgrind [81]	×	×	✓	✓	×	×
Valgrind [39]	×	×	✓	✓	×	×
Open Source?	N	Y	Y	Y	Y	Y
Determine Thread Model	×	×	×	×	×	×
Detect Deadlock	×	✓	✓	✓	✓	×
Detect Datarace	×	×	P	P	P	×
Detect Potential Livelock	×	×	×	×	×	×
Memory Leak Detection	✓	✓	✓	✓	✓	✓
Memory Overflow Detection	P	×	P	P	P	×
Optional GC	×	×	×	×	×	×
Dynamic Aspect Injection	×	×	×	×	×	P

✓- Feature Present × - Feature Absent P - Partial

negative in the leak detector as our garbage collector failed to capture a free function which was called through a wrapper in certain scenarios. However, all of these cases are related to individual tools and are not caused by the integration implemented in ParallelC-Assist.

V. CONCLUSION

We build an integrated architecture – ParallelC-Assist to support developers in maintaining multi-threaded applications in C through the detection of concurrency-related bugs, analysis of concurrency-related design aspects, memory management, and logging facilities. The architecture was built using the dynamic instrumentation framework of `PIN` [37] and re-targets its interconnection APIs for integration with various IDEs and Debuggers. Thus, ParallelC-Assist provides easy-to-use interfaces, and we demonstrate a prototype integration with Eclipse CDT. Further, ParallelC-Assist provides a framework to write various other analysis tools according to the developer's requirement to comprehend any aspect of a C code. We set up the architecture with an initial tool set for debugging (deadlock, data race, and livelock), extracting concurrency-related design elements based on thread-resource interaction, automated garbage collection, and dynamic code weaving. We tested the integrated architecture over the pthread CDAC [25] benchmark and achieved overall precision and recall scores of 98.13% and 98.56%, respectively. We study readily available tools integrated with the popular plug-ins and compared them in terms of the offered features listed in Table 5. `Visual Studio Profiler` provides many of the features supported by ParallelC-Assist; however, it is commercial, does

not support code injection, and does not provide a framework to write and customizing new tools. We intend to extend our tool in the future as follows:

- Extend ParallelC-Assist with other IDEs such as `Visual Studio`, `Code Blocks`, and other debuggers such as `Microsoft Visual Studio` debugger, etc.
- Analyse the utility of a compatible ParallelC-Assist over languages like C++ or Rust.

REFERENCES

[1] S. M. H. Dehaghani and N. Hajrahimi, "Which factors affect software projects maintenance cost more?" *Acta Inf. Medica*, vol. 21, no. 1, pp. 63–72, 2013.

[2] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Prof.*, vol. 2, no. 3, pp. 17–23, 2000.

[3] J. Koskinen, "Software maintenance costs," Inf. Technol. Res. Inst., Univ. Jyväskylä, Jyväskylä, Finland, Tech. Rep., 2015.

[4] H. Krasner, "The cost of poor quality software in the US: A 2018 report," Consortium for IT Software Quality (CISQ), Needham, MA, USA, Tech. Rep., 2018.

[5] L. Prechelt, "An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl," *IEEE Comput.*, vol. 33, no. 10, pp. 23–29, Mar. 2000.

[6] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[7] S.-E. Choi and E. C. Lewis, "A study of common pitfalls in simple multi-threaded programs," in *Proc. 31st SIGCSE Tech. Symp. Comput. Sci. Educ.*, Mar. 2000, pp. 325–329.

[8] C.-P. Chen, "The parallel debugging architecture in the Intel debugger," in *Proc. Int. Conf. Parallel Comput. Technol.* Cham, Switzerland: Springer, 2003, pp. 444–451.

[9] *Intel Inspector User Guide for Linux*OS*. Accessed: May 3, 2020. [Online]. Available: https://software.intel.com/en-us/inspector-user-guide-linux-data-race

[10] G. Zitzlsberger. *Using Intel C++ Compiler With the Eclipse* IDE on Linux*. Accessed: May 3, 2020. [Online]. Available: https://software.intel.com/

[11] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "ConSeq: Detecting concurrency bugs through sequential errors," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 39, 2011, pp. 251–264.

[12] N. Chatterjee, S. Majumdar, S. R. Sahoo, and P. P. Das, "Debugging multi-threaded applications using pin-augmented GDB (PGDB)," in *Proc. Int. Conf. Softw. Eng. Res. Pract.*, 2015, pp. 109–115.

[13] Y. Cai and L. Cao, "Effective and precise dynamic detection of hidden races for Java programs," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, Aug. 2015, pp. 450–461.

[14] Microsoft. *Microsoft Concurrency Visualizer*. Accessed: Feb. 1, 2019. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd537632.aspx

[15] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multi-threaded software systems by using trace visualization," in *Proc. 5th Int. Symp. Softw. Visualizat.*, Oct. 2010, pp. 133–142.

[16] M. Jain and D. Gopalani, "Use of aspects for testing software applications," in *Proc. IEEE Int. Advance Comput. Conf. (IACC)*, Jun. 2015, pp. 282–285.

[17] S. Iqbal and G. Allen, "Representing aspects in design," in *Proc. 3rd IEEE Int. Symp. Theor. Aspects Softw. Eng.*, Jul. 2009, pp. 313–314.

[18] *Intel Parallel Stdio XE*. Accessed: May 3, 2020. [Online]. Available: https://software.intel.com/en-us/parallel-studio-xe

[19] *Intel System Studio*. Accessed: May 3, 2020. [Online]. Available: https://software.intel.com/en-us/system-studio

[20] W. Keller, "International technology diffusion," *J. Econ. Literature*, vol. 42, no. 3, pp. 752–782, 2004.

[21] S. J. Vaughan-Nichols, "Building better software with better tools," *Computer*, vol. 36, no. 9, pp. 12–14, Sep. 2003.

[22] S. Majumdar, N. Chatterjee, S. R. Sahoo, and P. P. Das, "D-Cube: Tool for dynamic design discovery from multi-threaded applications using PIN," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Aug. 2016, pp. 25–32.

[23] N. Chatterjee, S. Bose, and P. P. Das, "Dynamic weaving of ASPECTs in C/C++ using PIN," in *Proc. Int. Conf. High Perform. Compilation, Comput. Commun.*, Mar. 2017, pp. 55–59.

[24] N. Chatterjee, S. S. Thakur, and P. P. Das, "Resource management in native languages using dynamic binary instrumentation (PIN)," in *Advanced Computing and Systems for Security*. Cham, Switzerland: Springer, 2016, pp. 107–119.

[25] CDAC. *In House Pthreads Benchmarks*. Accessed: Feb. 1, 2019. [Online]. Available: https://www.cdac.in/index.aspx?id=ev_hpc_hypack_pthreads_overview

[26] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind$^+$: An efficient dynamic race detector," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–13.

[27] GNU Free Software Foundation. (2017). *GDB: The GNU Project Debugger*. [Online]. Available: http://www.gnu.org/software/gdb

[28] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger canvas: Industrial experience with the code bubbles paradigm," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 1064–1073.

[29] S. Chakraborty and V. Vafeiadis, "Validating optimizations of concurrent C/C++ programs," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2016, pp. 216–226.

[30] Microsoft. *Visual Studio Profiler*. Accessed: May 3, 2020. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc337887.aspx

[31] A. K. Kolawa and C. E. Byers, "Modularizing a computer program for testing and debugging," U.S. Patent 6 895 578, May 17, 2005.

[32] Valgrind Developers. (2021). *Memcheck: A Memory Error Detector*. [Online]. Available: https://valgrind.org/docs/manual/mc-manual.html

[33] Y. Chen, Y.-H. Lee, W. E. Wong, and D. Guo, "A race condition graph for concurrent program behavior," in *Proc. 3rd Int. Conf. Intell. Syst. Knowl. Eng.*, vol. 1, Nov. 2008, pp. 662–667.

[34] Y. W. Song and Y. Lee, "Efficient data race detection for C/C++ programs using dynamic granularity," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 679–688.

[35] M. Christiaens and K. De Bosschere, "Accordion clocks: Logical clocks for data race detection," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2001, pp. 494–503.

[36] M. Moiseev, M. Glukhikh, A. Zakharov, and H. Richter, "A static analysis approach to data race detection in SystemC designs," in *Proc. IEEE 16th Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2013, pp. 54–59.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "PIN: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Notice*, 2005, pp. 190–200.

[38] (2017). Valgrind Developers. *Valgrind*. [Online]. Available: http://valgrind.org/

[39] M. Rai. (2008). *Memory Leak Detection Using Windbg*. [Online]. Available: https://www.codeproject.com

[40] M. Pool. (2022). *CCMALLOC*. [Online]. Available: http://cs.ecs.baylor.edu/~donahoo/tools/ccmalloc/

[41] F. Germain. (2011). *LeakTracer—Trace and Analyze Memory Leaks in C++ Programs*. [Online]. Available: http://www.andreasen.org/leaktracer/

[42] Yurikovitch. (2013). *MEMDebug*. [Online]. Available: https://sourceforge.net/projects/memdebug/

[43] J. Belmonte, P. Dugerdil, and A. Agrawal, "A three-layer model of source code comprehension," in *Proc. Indian Softw. Eng. Conf. (ISEC)*, 2014, pp. 10–14.

[44] M. Mirakhorli and J. Cleland-Huang, "Detecting, tracing, and monitoring architectural tactics in code," *IEEE Trans. Softw. Eng.*, vol. 42, no. 3, pp. 205–220, Mar. 2016.

[45] D. Djuric and V. Devedzic, "Incorporating the ontology paradigm into software engineering: Enhancing domain-driven programming in Clojure/Java," *IEEE Trans. Syst., Man, Cybern., C, Appl. Rev.*, vol. 42, no. 1, pp. 3–14, Jan. 2012.

[46] K. Brown, "Design reverse-engineering and automated design-pattern detection in smalltalk," North Carolina State Univ., Raleigh, NC, USA, Tech. Rep., 1996.

[47] H. Lee, H. Youn, and E. Lee, "Automatic detection of design pattern for reverse engineering," in *Proc. 5th ACIS Int. Conf. Softw. Eng. Res., Manage. Appl. (SERA)*, Aug. 2007, pp. 577–583.

[48] G. Antoniol and Y.-G. Gueheneuc, "Feature identification: An epidemiological metaphor," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 627–641, Sep. 2006.
[49] S. P. Reiss, "Visualizing program execution using user abstractions," in *Proc. ACM Symp. Softw. Visualizat.*, 2006, pp. 125–134.
[50] J. Quante and R. Koschke, "Dynamic protocol recovery," in *Proc. 14th Work. Conf. Reverse Eng.*, Oct. 2007, pp. 219–228.
[51] N. R. Tallent and J. M. Mellor-Crummey, "Effective performance measurement and analysis of multithreaded applications," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 229–240, Feb. 2009.
[52] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An aspect-oriented extension to the C++ programming language," in *Proc. Int. Conf. Tools Pacific, Objects Internet, Mobile Embedded Appl.*, 2002, pp. 53–60.
[53] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proc. Eur. Conf. Object-Oriented Program.* Cham, Switzerland: Springer, 2001, pp. 327–354.
[54] D. Geer, "Eclipse becomes the dominant Java IDE," *Computer*, vol. 38, no. 7, pp. 16–18, 2005.
[55] C. Griffin, "Introduction to the eclipse modeling framework," in *Proc. OMG MDA Implementer's Workshop*, 2003.
[56] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *Proc. Int. Conf. Comput. Aided Verification*. Cham, Switzerland: Springer, 2001, pp. 260–264.
[57] S. Blair-Chappell and A. Stokes, *Parallel Programming With Intel Parallel Studio XE.*. Hoboken, NJ, USA: Wiley, 2012.
[58] A. Kleen and B. Strong, "Intel processor trace on Linux," *Tracing Summit*, vol. 1, pp. 1–18, Aug. 2015.
[59] Intel. *Pin 3.2 User Guide*. Accessed: Apr. 25, 2020. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/
[60] QuarksLab. *A Dynamic Binary Instrumentation Framework Based on LLVM*. Accessed: Apr. 25, 2020. [Online]. Available: https://github.com/QBDI/QBDI
[61] Q. Wang, H. Shu, Y. Li, and H.-J. Huang, "Malicious code behavior analysis based on dynamorio," *Comput. Eng.*, vol. 37, p. 18, Jan. 2011.
[62] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.
[63] A. Almomany, A. Alquraan, and L. Balachandran, "GCC vs. ICC comparison using PARSEC benchmarks," *Int. J. Innov. Technol. Exploring Eng.*, vol. 4, no. 7, pp. 1–13, 2014.
[64] IBM. (2020). *Eclipse CDT (C/C++ Development Tooling)*. Eclipse Foundation. [Online]. Available: https://www.eclipse.org/cdt/
[65] J. Reinders, *VTune Performance Analyzer Essentials*. Mountain View, CA, USA: Intel Press, 2005.
[66] *Software Verify*. Accessed: Feb. 1, 2021. [Online]. Available: https://www.softwareverify.com/contact-software-verification.php
[67] The LLDB Team. (2020). *The LLDB Debugger*. [Online]. Available: https://lldb.llvm.org/
[68] A. K. Ghoshal, N. Chatterjee, A. Chakrabarti, and P. Das, "Design of PIN-augmented debugger for multi-threaded applications," in *Innovations in Computer Science and Engineering*, May 2018, pp. 153–159.
[69] N. Chatterjee, S. Thakur, and P. P. Das, "Resource management in native languages using dynamic binary instrumentation (PIN)," in *Proc. 2nd Int. Doctoral Symp. Appl. Comput. Secur. Syst. (ACSS)*, 2015, p. 107.
[70] Microsoft. (2017). *Debugging in Visual Studio*. Microsoft Developer Network. [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/sc65sadd.aspx
[71] R. Stanek. (2020). *Apache Netbeans*. Apache. [Online]. Available: https://netbeans.org/
[72] MortenMacFly. (2020). *Code: Blocks—The Open Source, Cross Platform, Free C, C++ and Fortran IDE*. [Online]. Available: http://www.codeblocks.org/
[73] R. M. Albrecht. (Aug. 2012). *IDB: Intel Debugger*. Intel Software Developer Zone. [Online]. Available: http://software.intel.com/en-us/articles/idb-linux
[74] IBM. (2019). *IBM Rational Rose Enterprise 7.0.0.4 IFIX001*. IBM Rational Rose XDE. [Online]. Available: https://www.ibm.com/support/pages/ibm-rational-rose-enterprise-7004-ifix001
[75] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 355–372, Nov. 2013.
[76] CodePlex. (2017). *Visual Leak Detector for Visual C++ 2008–2015*. [Online]. Available: https://vld.codeplex.com/
[77] Free Software Foundation. (2014). *GNU Checker*. [Online]. Available: https://www.gnu.org/software/checker/checker.html
[78] The Clang Team. (2020). *Free Software Foundation*. [Online]. Available: https://clang.llvm.org/docs/ThreadSanitizer.html
[79] Valgrind Developers. (2019). *Helgrind: A Thread Error Detector*. [Online]. Available: https://valgrind.org/docs/manual/hg-manual.html

NACHIKETA CHATTERJEE received the B.Tech. degree in information technology from the University of Calcutta, West Bengal, India, in 2004, where he is currently pursuing the Ph.D. degree with the A. K. Choudhury School of Information Technology. He has been a Consultant with Tata Consultancy Services Ltd., Kolkata, India, since 2006. He was with Skytech Solutions Pvt. Ltd., India, for 2.5 years in the area of application development for airlines and retail domain. His main research interest includes improve the software development process with efficient productivity tools, focusing on profiling and analytics. He has received the Best Performance Improvement and Innovation Pride Award from TCS for accelerating the process with improved tool strategy for faster time to market for world's second DIY retailer, in 2020 and 2021.

SRIJONI MAJUMDAR (Student Member, IEEE) received the Ph.D. degree in the area of program analysis and knowledge mining using machine learning frameworks from the Advanced Technology Development Centre, Indian Institute of Technology, Kharagpur, India. She was with Tata Consultancy Services Ltd., Mumbai, India, in the area of performance engineering of software systems and data analytics. She is currently a Postdoctoral Researcher with the School of Computing, University of Leeds, and work in the area of computational social sciences. She is actively involved with several developers from the software industry for her research on software maintenance. Her main research interest includes software maintenance, focusing on building knowledge mining systems from source code and related metadata (big code). She is an Executive Member of the IEEE Women in Engineering, Asia Pacific Kharagpur Branch. More information is available at https://sites.google.com/site/srijonicse/home.

PARTHA PRATIM DAS (Member, IEEE) received the B.Tech., M.Tech., and Ph.D. degrees from the Indian Institute of Technology Kharagpur (IIT Kharagpur), India, in 1984, 1985, and 1988, respectively.

He was a Faculty Member with the Department of Computer Science and Engineering, IIT Kharagpur, from 1988 to 1998. In 1998, he moved to the industry and was in director positions, until 2011. He is currently a Professor with the Department of Computer Science and Engineering, IIT Kharagpur. He is also on long leave from IIT Kharagpur and a Visiting Professor with Ashoka University, India. He was the Joint Principal Investigator of the National Digital Library of India Project of the Ministry of Education, Government. of India, from 2015 to 2022, and led the initiative to integrate the digital repositories of various institutions and publishers across India. He has published more than 100 papers in national and international journals and conferences. His current interests include software productivity and quality, human–computer interaction, computer analysis of Indian classical dance, and technology-enhanced learning.

Dr. Das has received several recognitions, including the UNESCO/ROSTSCA Young Scientist in 1989, the INSA Young Scientist Award in 1990, the Young Associateship of the Indian Academy of Sciences, in 1992, the UGC Young Teachers' Career Award, in 1993, the INAE Young Engineer Award, in 1996, the Interra Special (Process) Recognition, in 2009, and the Interra 10 Years' Tenure Plaque, in 2011. He was a co-recipient of the mBillionth Awards by the Digital Empowerment Foundation, in 2017, the Gems of Digital India Award, in 2019, and the Open Education Award for Excellence in Open Resilience Category for the National Digital Library of India, in 2020. He is also the Editor-in-Chief of the *Journal of the Institution of Engineers: Series B*.

AMLAN CHAKRABARTI (Senior Member, IEEE) received the M.Tech. degree from the University of Calcutta and the Ph.D. degree from the Indian Statistical Institute, Kolkata. He was a Postdoctoral Fellow with the School of Engineering, Princeton University, USA, from 2011 to 2012. He is currently a Professor with the A. K. Choudhury School of Information Technology, University of Calcutta. His research interests include machine learning, computer vision, cyber-physical systems, reconfigurable computing, quantum computing, and VLSI CAD. He is a Senior Member of ACM, the IEEE Computer Society Distinguished Visitor (2020–2022), the Distinguished Speaker of ACM, the Secretary of IEEE CEDA India Chapter, the Vice President of the Society for Data Science, and a Life Member of CSI India. He was a recipient of the DST BOYSCAST Fellowship Award in Engineering Science, in 2011, the Indian National Science Academy (INSA) Visiting Faculty Fellowship, in 2014, the JSPS Invitation Research Award, in 2016, the Erasmus Mundus Leaders Award from European Union, in 2017, and the Hamied Visiting Professorship from the University of Cambridge, U.K., in 2018. He is the Series Editor of the *Transactions on Computer Systems and Networking* (Springer), an Associate Editor of *Journal of Computers and Electrical Engineering* (Elsevier), and the Guest Editor of *Journal of Applied Sciences* (Springer).