Dynamic and Static Code Analysis for Java Programs on Heterogeneous Hardware

Athanasios Stratikopoulos
The University of Manchester
United Kingdom

Tianyu Zuo*
CCB Fintech Co., Ltd.
China

Umut Sarp Harbalioglu
The University of Manchester
United Kingdom

Juan Fumero
The University of Manchester
United Kingdom

Michail Papadimitriou
The University of Manchester
United Kingdom

Orion Papadakis
The University of Manchester
United Kingdom

Maria Xekalaki
The University of Manchester
United Kingdom

Christos Kotselidis
The University of Manchester
United Kingdom

Abstract

The increasing prevalence of heterogeneous computing systems, incorporating accelerators like GPUs, has spurred the development of advanced frameworks to bring high performance capabilities to managed languages. TornadoVM is a state-of-the-art, open-source framework for accelerating Java programs. It enables Java applications to offload computation onto GPUs and other accelerators, thereby bridging the gap between the high-level abstractions of the Java Virtual Machine (JVM) and the low-level, performance-oriented world of parallel programming models, such as OpenCL and CUDA. However, this bridging comes with inherent trade-offs. The semantic and operational mismatch between these two worlds - such as managed memory versus explicit memory control, or dynamic JIT compilation versus static kernel generation - TornadoVM to limit or exclude support for certain Java features. These limitations can hinder developer productivity and make it difficult to identify and resolve compatibility issues during development.

This paper introduces `TornadoInsight`, a tool that simplifies development with TornadoVM by detecting incompatible Java constructs through static and dynamic analysis. `TornadoInsight` is developed as an open-source IntelliJ IDEA plugin that provides immediate, source-linked feedback within the developer's workflow. We present the architecture of `TornadoInsight`, detail its inspection mechanisms, and evaluate its effectiveness in improving the development workflow for TornadoVM users. `TornadoInsight` is publicly available and offers a practical solution for enhancing developer experience and productivity in heterogeneous managed runtime environments.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**.

Keywords: Tool, Program Analysis, Java, Heterogeneous Hardware

ACM Reference Format:
Athanasios Stratikopoulos, Tianyu Zuo, Umut Sarp Harbalioglu, Juan Fumero, Michail Papadimitriou, Orion Papadakis, Maria Xekalaki, and Christos Kotselidis. 2025. Dynamic and Static Code Analysis for Java Programs on Heterogeneous Hardware. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3759426.3760984

1 Introduction

Modern computing architectures have become heterogeneous, leveraging specialized hardware accelerators such as GPUs and FPGAs, to boost performance for parallel workloads. While low-level programming models, such as OpenCL [9] and CUDA [3], provide fine-grained control over hardware accelerators, they present a steep learning curve - particularly for developers accustomed to high-level, object-oriented languages like Java and C++. This challenge is even more pronounced for managed languages such as Java, where features like automatic memory management, strong type safety, and runtime introspection are fundamentally at odds with the explicit memory handling, weak typing, and rigid execution models of the low-level programming models. Bridging these paradigms requires not only rethinking programming abstractions but also navigating semantic mismatches that complicate integration, debugging, and performance tuning. TornadoVM has emerged as a promising solution, enabling Java applications to offload computations

*This work was carried out while Tianyu Zuo was completing his Master's degree at The University of Manchester.

MPLR '25, October 12–18, 2025, Singapore, Singapore

ACM ISBN 979-8-4007-2149-6/25/10.
https://doi.org/10.1145/3759426.3760984

onto hardware accelerators without requiring developers to write low-level accelerator code [1, 4, 5, 11].

Despite its advancements, programming with TornadoVM presents its own set of challenges. A primary hurdle is that TornadoVM supports a subset of Java features due to differences in programming models between Java and underlying accelerator technologies (e.g., OpenCL [9], CUDA [3], SPIR-V [6]). For example, exception handling constructs - such as traps and catch clauses - are unsupported within accelerated code regions (e.g., methods), as the underlying OpenCL and CUDA drivers lack the capability to propagate or manage such runtime behaviors. Similarly, dynamic memory allocation is not permitted in these regions because the generated accelerated code is statically compiled; hence, any attempt to introduce dynamic constructs would necessitate recompilation, disrupting performance and correctness. These limitations are not always apparent to developers, resulting in runtime errors or unexpected behavior due to the inadvertent usage of unsupported features. Furthermore, TornadoVM's exception handling can be obscure, making it difficult for developers to pinpoint the source of issues. Consequently, there is a pressing need for a developer-centric tool that can identify these incompatibilities early in the development cycle and provide actionable feedback.

To address this need, we have developed `TornadoInsight`, a tool specifically designed to simplify development with TornadoVM and make heterogeneous computing more accessible to Java developers. Delivered as an open-source[1] IntelliJ IDEA plugin, `TornadoInsight` integrates naturally into the developer's workflow, offering immediate and actionable insights during code development. `TornadoInsight` enhances developer productivity by:

- Employing **static analysis** using IntelliJ IDEA's Program Structure Interface (PSI) to detect common TornadoVM incompatible Java features directly as code is written.
- Introducing a novel **dynamic inspection module** that leverages the developer's installed TornadoVM software, to enable the detection of issues that are not identifiable through static analysis alone, such as dynamic memory allocation patterns manifesting at runtime.
- Providing **clear visual feedback** within the IDE, highlighting problematic code sections and offering explanations for the incompatibility.
- Offering a **user-friendly interface**, including a tool window for managing inspections and a dedicated console for dynamic analysis output.

2 Background

2.1 TornadoVM and Programming Challenges

TornadoVM allows Java developers to offload the execution of parts of their applications on heterogeneous hardware, by dynamically compiling Java bytecode to OpenCL C, PTX, or SPIR-V for execution on devices, such as multi-core CPUs, GPUs, and FPGAs [1, 5]. It uses a task-based model, where each task is annotated with TornadoVM annotations, such as `@Parallel` and `@Reduce`, which are used by developers to express parallelizable loops and reduction operations, respectively [4, 10]. Tasks with interdependencies—such as one producing data consumed by another—can be composed within a `TaskGraph`, enabling the formation of task chains. This structure allows TornadoVM to analyze and optimize the data flow across tasks, potentially improving the transfer efficiency between the host CPU and accelerator devices.

Despite offering a hardware-agnostic Java API, TornadoVM presents challenges for Java developers due to the semantic mismatch between the high-level abstractions of the JVM, and the low-level performance-centric nature of parallel programming models, such as OpenCL and CUDA. This trade-off primarily arises from the following constraints:

- **No Object Support:** Instantiating arbitrary objects on accelerators is not supported due to differences in terms of the memory layout and management between Java and C-based parallel programming models [13]. Only primitive types, their arrays, and specific TornadoVM-provided objects (e.g., *VectorFloat*) are supported by the TornadoVM compiler. For instance, the TornadoVM *VectorFloat4* object is compiled by TornadoVM to use OpenCL vector types (*float4*).
- **No Recursion:** Heterogeneous parallel programming models do not support or have limited support (CUDA) of recursion due to assumptions of deterministic execution and limited stack space on GPUs.
- **Limited Exception Handling:** Exception handling on GPUs is fundamentally limited. For instance, in a division-by-zero scenario on the CPU, a flag is typically set in a special register, which the operating system can then inspect and relay to the application runtime—such as the Java Virtual Machine (JVM)—to properly handle the exception. In contrast, GPUs lack such exception propagation mechanisms, meaning there is no standard way to signal or manage exceptions at the hardware or driver level [2]. As a result, TornadoVM must insert additional control-flow constructs to ensure that such exceptional conditions are preemptively avoided during execution.
- **No Static TaskGraphs/Tasks:** Defining `TaskGraph` objects or tasks as static fields can lead to deadlocks between the user thread running class initialization and the TornadoVM JIT compiler; hence it is not supported.
- **No Dynamic Memory Allocation:** Heterogeneous parallel programming models, such as OpenCL and CUDA, do not support dynamic allocation of buffers used for data transfer between the CPU and accelerator devices. In adherence to this constraint, TornadoVM offers automated memory management by abstracting the allocation

[1] GitHub link: https://github.com/beehive-lab/tornado-insight

and deallocation of buffers for each task. As a result, the dynamic memory allocation at runtime is not permitted within the TornadoVM execution model.

- **No Invocation of JVM or Native Libraries:** Tasks compiled for execution on accelerator devices are restricted from invoking functions that interact with the JVM, native libraries, or the operating system - such as I/O operations, reflection, or thread management - due to their execution context being isolated from the host environment. However, certain exceptions exist. For instance, calls to mathematical functions (e.g., sin, cos) from the `java.lang.Math` package are recognized by TornadoVM's compiler and replaced with equivalent low-level intrinsics in the target backend (e.g., OpenCL, PTX, or SPIR-V).

Due to these constraints, developers often learn which Java features are unsupported only after encountering obscure console output.

2.2 IntelliJ Platform SDK

`TornadoInsight` is built as an IntelliJ IDEA plugin, leveraging the **IntelliJ Platform SDK** [8]. The SDK provides a rich environment for building developer tools. To implement `TornadoInsight`, we used the following key components:

- **Program Structure Interface (PSI):** PSI is crucial for static analysis, as it parses files and creates a syntactic and semantic model of the code. Various PSI elements (e.g., classes, methods, variables) are organized hierarchically, allowing for detailed traversal and code inspection.
- **Extension Points:** The IntelliJ Platform allows plugins to extend the functionality of the editor through declared extension points. `TornadoInsight` uses these for creating custom code inspections, tool windows, and settings panels, facilitating its seamless integration into the IDE.
- **Java Swing:** This library is useful for the development of the user interface components, such as the tool windows and dialogs.

3 `TornadoInsight` Architecture

`TornadoInsight` is architected to provide a seamless and informative experience for developers using TornadoVM. This section presents the core components of `TornadoInsight`, including a static inspection module (Section 3.1), a dynamic inspection module (Section 3.2), a user interface module (Section 3.3), and a communication module (Section 3.4) that coordinates their activities. A prime design principle has been to maximize efficiency by relying heavily on static checks, while also reserving more resource-intensive dynamic checks for issues that cannot be identified statically.

3.1 Static Inspection Module

The static inspection module leverages the JetBrains code inspection framework [7] to perform on-the-fly checks as the developer writes code. Custom inspections are implemented by extending the `AbstractBaseJavaLocalInspectionTool` Java class, and providing a `PsiElementVisitor` (typically a `JavaElementVisitor` or `JavaRecursiveElementVisitor`) to traverse the PSI tree of the active Java file. When a TornadoVM annotation, such as `@Parallel` or `@Reduce` is found, its parent method (i.e., the TornadoVM task) is identified. The task body is then traversed to apply specific inspection rules. Any detected issues are reported via a `ProblemsHolder`, which serves as a mechanism for registering problems (e.g., warnings, errors, or informational messages). Thus, `TornadoInsight` can highlight the problematic code and display a descriptive message.

3.1.1 Implemented Inspectors.

`TornadoInsight` includes six distinct types of inspectors, each classified according to its functional role in identifying TornadoVM-incompatible constructs:

1. **Data Type Inspector:** Verifies that method parameters and local variables within TornadoVM tasks conform to a set of supported primitive types, their array equivalents, and recognized TornadoVM-specific data structures (e.g., `VectorFloat`). The list of supported types is deserialized from a configuration file to improve maintainability and extensibility.
2. **Recursion Inspector:** Analyzes method call expressions within a task to detect both direct recursion (where a task method calls itself) and indirect recursion (where a sequence of method calls within the same compilation unit eventually leads back to the original task method).
3. **Exceptions Inspector:** Identifies the presence of exception-related constructs, such as *throw* statements, *try-catch* blocks, and methods declaring *throw* clauses; which are generally unsupported in code compiled by TornadoVM.
4. **Assert Inspector:** Flags the use of *assert* statements within tasks, as these may be ignored during TornadoVM compilation and execution.
5. **Static Task & TaskGraph Inspector:** Detects improper declarations of `TaskGraph` fields or task definitions within `static` initializer blocks, which are disallowed under the TornadoVM execution model.
6. **JVM, OS-dependent, and Native Method Call Inspector:** Flags invocations of JVM or OS-dependent methods within tasks, including references to known unsupported classes (e.g., `java.lang.System`, etc.) and any methods marked as native.

3.2 Dynamic Inspection Module

To catch issues missed by static analysis (e.g., code generation or dynamic memory usage), `TornadoInsight` includes a dynamic inspection module. Developers can select a statically verified TornadoVM task from the tool window, trigger

JIT compilation, and provide input parameters via an interactive dialog. Upon user configuration, `TornadoInsight` performs the following steps:

1. **Generates a Java class:** At runtime, a new Java class is automatically generated containing the selected task, a `main` method to initialize input parameters and the associated `TaskGraph`, along with all necessary imports. The method parameters are encapsulated using a `Method` entity.
2. **Compiles the generated class:** The generated class is compiled using `javac`, with the TornadoVM runtime JAR included in the classpath.
3. **Packages into an executable JAR:** A manifest file specifying the entry point is created, and the compiled class (.class) is packaged into an executable JAR archive.
4. **Executes using TornadoVM:** The generated JAR is executed within the user's configured TornadoVM environment. TornadoVM is configured by the users in the settings panel (see Section 3.3).
5. **Analyzes runtime output:** The standard output from the TornadoVM runtime is captured and analyzed for specific exceptions or known error patterns. For example, the inspector can detect runtime failures related to unsupported dynamic array creation.

This entire process runs on a separate thread to avoid blocking the user interface.

3.3 User Interface and Developer Experience

`TornadoInsight` features a user interface designed to provide a non-intrusive yet informative experience, seamlessly integrated into the IntelliJ IDEA environment.

- **Settings Panel:** Embedded within the IntelliJ settings interface, this panel enables users to specify the path to their TornadoVM environment variable file (Figure 1). The plugin verifies the TornadoVM setup for correctness and completeness, via the *Configuration Listener* (see Section 3.4).
- **Tool Window:** A dedicated *TornadoVM* tool window lists the valid TornadoVM tasks identified in the open Java class file, as shown in Figure 2a. It also displays the result of the static inspection, and allows users to initiate the dynamic inspection process by selecting a task from the list.
- **Real-time Static Feedback:** Results from static analysis are presented as inline code annotations, such as red wavy underlines, with tooltips offering concise explanations. Where applicable, detailed descriptions or documentation links are provided to guide users in resolving compatibility issues, as presented in Figure 2b.
- **Console Output:** The *TornadoInsight Console* provides real-time feedback on the progress and results of dynamic inspections, allowing users to monitor the execution and debug output directly within the IDE. For instance, Figure 3 shows an OpenCL C kernel for vector addition successfully generated by TornadoVM during dynamic inspection. `TornadoInsight` used the user-defined array size as defined in the *Settings Panel* and reports the overall inspection time.

3.4 Communication and Data Synchronization

To ensure a responsive and seamless user experience within `TornadoInsight`, requires effective communication between its code components. This is accomplished through a combination of event listeners (Section 3.4.1) and a robust data synchronization mechanism (Section 3.4.2).

3.4.1 Event Listeners. `TornadoInsight` leverages several IntelliJ Platform listeners to monitor and respond to changes in the development environment:

- **Configuration Listener**: Triggered when a project is opened, the `ProjectManager` listener checks whether the TornadoVM environment has been properly configured. If not, the user is notified with a corresponding message.
- **File Activity Listeners**: The `Tool Window` and `FileEditorManager` listeners respond to tool window activation, file switches, and changes within the currently open file. This ensures that the task list displayed in the tool window remains accurate and up to date as the user navigates and modifies source files.
- **Live Code Edit Listener**: `TornadoInsight`employs the `PsiTreeChangeListener` to monitor real-time modifications in the Program Structure Interface tree, allowing for immediate updates to the task list as the user modifies code within the active file.

3.4.2 Data Synchronization. To ensure correctness, the tool window displays only those tasks that are both syntactically valid and free from static analysis violations. When a static inspector identifies a violation and registers it using `ProblemsHolder`, the corresponding `PsiMethod` is added to a shared, thread-safe `CopyOnWriteArraySet` Java class. A custom message bus `Topic` (e.g., `TornadoTaskRefreshListener`) is then used to broadcast updates. Upon receiving such a message, the tool window component refreshes its task list and filters out any methods flagged as problematic.

4 Tool Demonstration

To demonstrate the usability and effectiveness of our tool, we describe a series of annotated screenshots demonstrating its key capabilities. These visual examples highlight how the plugin assists developers in identifying and resolving TornadoVM compatibility issues through static and dynamic analysis features integrated into IntelliJ IDEA.

4.1 Configuration Panel

`TornadoInsight` provides a dedicated settings panel where developers configure the plugin's behavior. Figure 1 presents

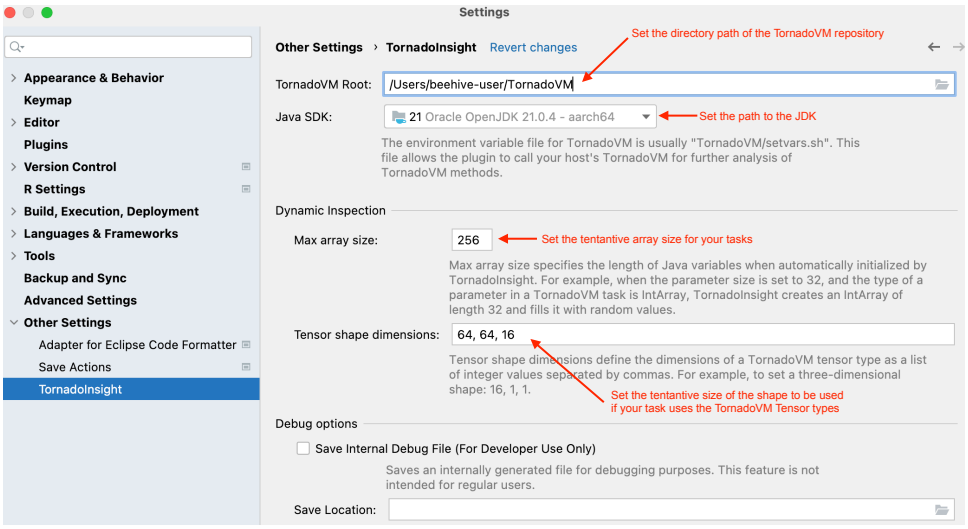
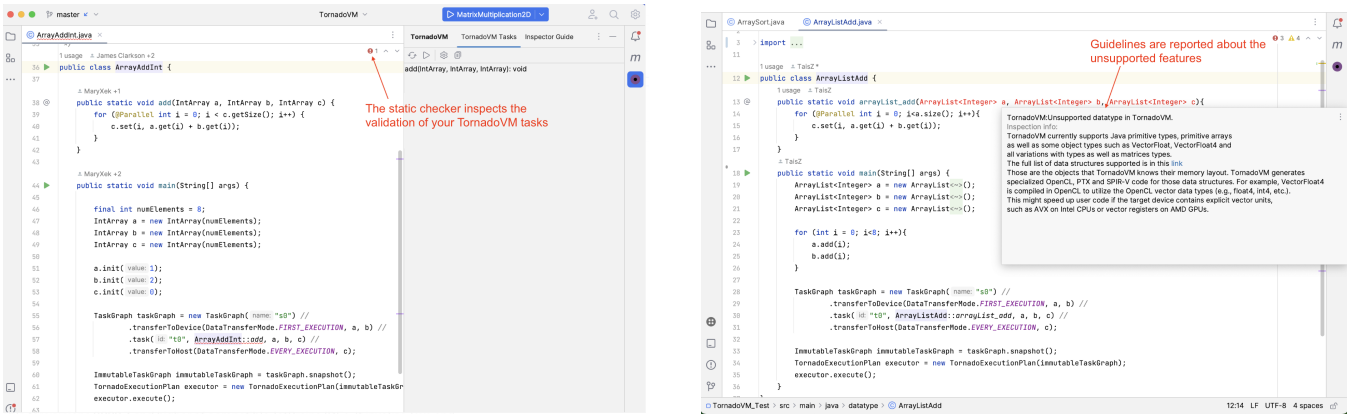


Figure 1. Overview of the `TornadoInsight` Settings Panel.



(a) Static Inspection and TornadoVM Tasks Window.

(b) Static Feedback of Unsupported Behaviour.

Figure 2. Overview of the TornadoInsight Settings and Static Inspection Workflow.

the settings panel that includes the configuration of the TornadoVM root directory, the Java SDK, default array sizes for dynamic analysis, and debug options. The debug options are intended for the plugin developers in order to log the automatically generated class, as discussed earlier in Section 3.2.

4.2 Static Analysis of TornadoVM Tasks

As the user writes Java code using TornadoVM annotations (e.g., `@Parallel`), the static checker actively inspects each task for unsupported constructs. Figure 2a shows an example that recognizes the TornadoVM annotation in line 39, and indicates that the static validation has detected an error via the red exclamation mark. The error is attributed to the *static* declaration of the task (line 57), which is not supported as described in Section 2.1. A second example of using the static analysis is shown in Figure 2b, where an unsupported data type, such as `ArrayList<Integer>` is used. In this case, the plugin provides a descriptive tooltip and guidance.

4.3 Dynamic Inspection Results

`TornadoInsight` enables runtime analysis of tasks by generating OpenCL kernels based on configured parameters. Users can invoke this from the tool window, and results are displayed in a dedicated console. When the inspected code is valid, `TornadoInsight` successfully runs the dynamic inspection and the generated kernel is displayed in the dedicated TornadoInsight Console (Figure 3), along with the inspection time. This immediate feedback enables developers to validate TornadoVM compatibility without leaving the IDE. When invalid constructs are present (e.g., unsupported method calls or exceptions), `TornadoInsight` reports the exact failure trace to help with debugging (Figure 4).

4.4 Usage and Integration of `TornadoInsight`

`TornadoInsight` is available on the JetBrains Marketplace and can be installed directly within IntelliJ IDEA (version

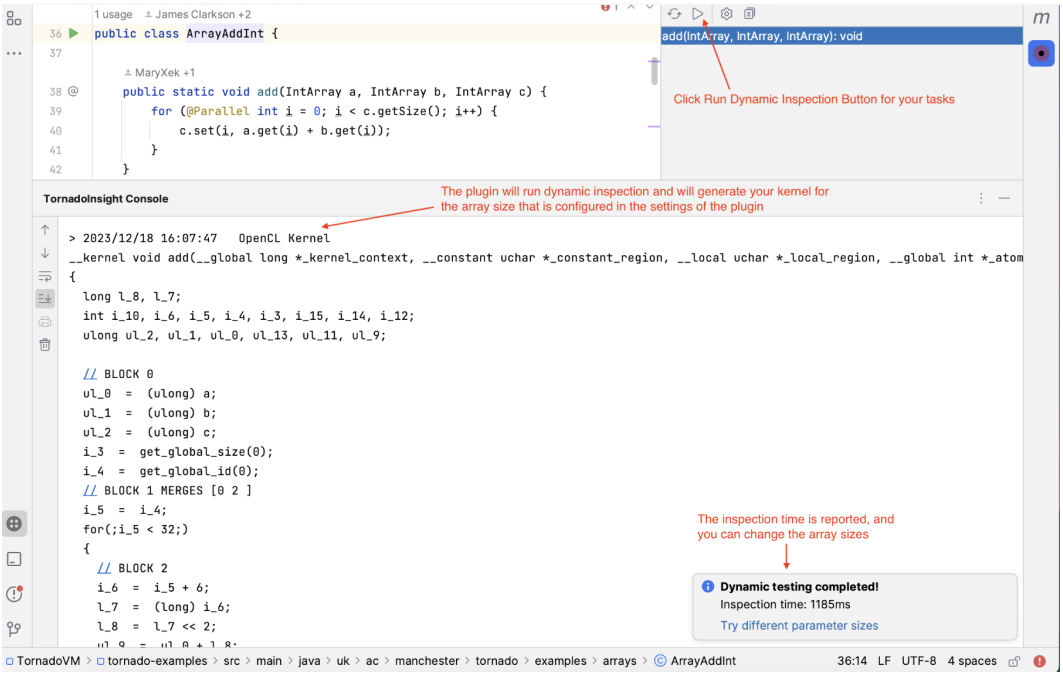


Figure 3. `TornadoInsight` Dynamic Inspection Output with Successful Behaviour.

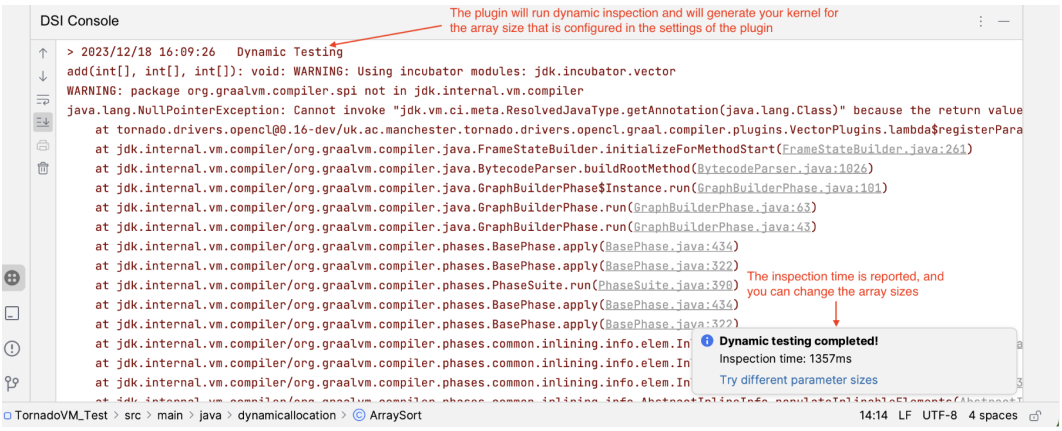


Figure 4. `TornadoInsight` Dynamic Inspection Output with Failed Behaviour.

2022.3 or later). Once installed, it can be used in any project that imports the TornadoVM API, enabling both static and dynamic inspections out of the box. This allows developers to catch compatibility issues early and optimize their workflow. Notable projects already benefiting from `TornadoInsight` are the TornadoVM-Ray-Tracer, GPULlama3.java [12], etc.

5 Conclusion and Future Work

This paper introduced `TornadoInsight` and demonstrated its practical use within IntelliJ IDEA through real-world examples. By combining static and dynamic analysis, `TornadoInsight` detects TornadoVM-incompatible Java constructs early. Our evaluation shows it gives actionable feedback that boosts productivity and lowers the barrier for Java development on heterogeneous hardware. As an open-source plugin, it promotes community use and contribution.

Future work includes focus on the precision of dynamic diagnostics and automated refactoring via IntelliJ's `QuickFix` API. These developments will reinforce `TornadoInsight` as a vital aid for Java developers working with hardware acceleration.

Acknowledgments

This work is supported by the European Union's Horizon Europe programme under grant agreements No 101070670 (ENCRYPT), No 101092850 (AERO), No 101093069 (P2CODE), and No 101070052 (TANGO). This work is also funded by UK

Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee (ENCRYPT: 10039809; AERO: 10048318; P2CODE: 10048316; TANGO: 10039107).

References

[1] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting high-performance heterogeneous hardware for Java programs using graal. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. doi:10.1145/3237009.3237016

[2] Nvidia Corporation. 2025. CUDA and Floating Point - Differences from x86. https://docs.nvidia.com/cuda/floating-point/index.html. Accessed: 2025-06-11.

[3] Nvidia Corporation. 2025. Programming Guide :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2025-06-11.

[4] Juan Fumero and Christos Kotselidis. 2018. Using Compiler Snippets to Exploit Parallelism on Heterogeneous Hardware: A Java Reduction Case Study *(VMIL 2018)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3281287.3281292

[5] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19)*. Association for Computing Machinery. doi:10.1145/3313808.3313819

[6] The Khronos® SPIR-V Working Group. 2025. SPIR-V Specification. Version 1.6: Unified. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html. Accessed: 2024-06-11.

[7] JetBrains. 2025. IntelliJ Platform Plugin SDK - JetBrains Code Inspections. https://plugins.jetbrains.com/docs/intellij/code-inspections.html. Accessed: 2025-06-11.

[8] JetBrains. 2025. JetBrains Plugin Documentation. https://plugins.jetbrains.com/docs/intellij/welcome.html. Accessed: 2025-06-11.

[9] Khronos Group. 2025. OpenCL. https://www.khronos.org/opencl/. Accessed: 2025-06-11.

[10] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. *SIGPLAN Not.* 52, 7 (April 2017), 74–82. doi:10.1145/3140607.3050764

[11] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin-Gabriel Blanaru, and Christos-Efthymios Kotselidis. 2021. Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21)*. Association for Computing Machinery, New York, NY, USA, 125–138. doi:10.1145/3453933.3454019

[12] Michail Papadimitriou, Mary Xekalaki, Juan Fumero, Athanasios Stratikopolos, Orion Papadakis, and Christos Kotselidis. 2025. *GPULlama3.java*. https://github.com/beehive-lab/GPULlama3.java

[13] Athanasios Stratikopoulos, Florin Blanaru, Juan Fumero, Maria Xekalaki, Orion Papadakis, and Christos Kotselidis. 2023. Cross-Language Interoperability of Heterogeneous Code *(Programming '23)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3594671.3594675

Received 2025-06-05; accepted 2025-07-28