**Efficient Implementation and Evaluation of Profilers in JavaScript-based Interpreters**

Kazuki Takehi

**Introduction:**

Profilers are essential tools in software development, providing crucial insights into program execution behavior and performance. This research focuses on implementing and evaluating efficient profiling techniques within a JavaScript-based interpreter for an original language. The single-threaded nature of JavaScript presents unique challenges, necessitating innovative approaches to profiling.

**Research Objectives:**

1. Implement and compare different profiling techniques in a JavaScript-based interpreter.
2. Overcome JavaScript's single-threaded limitations in statistical profiling.
3. Analyze trade-offs between accuracy and overhead for different profiling methods.

**Methodology:** Three profiling approaches were implemented and evaluated:

1. Event-based Profiler:

* Records start and end times of each function call.
* Provides accurate execution time measurements.
* Implements function entry and exit event capturing.

1. Statistical Profiler (Promise-based):

* Uses setInterval to queue sampling tasks every 1ms in the Task Queue.
* Implements frequent checkpoints in the program evaluation using await new Promise(...).
* At each checkpoint:
  + Program evaluation is temporarily paused.
  + Checks if a sampling task is in the Task Queue.
  + If present, records the currently executing function.
  + If not, resumes program evaluation.
* Challenges:
  + High overhead due to frequent pausing and resuming of program evaluation.
  + Potential for missing short-lived functions between checkpoints.

1. Statistical Profiler (Worker Thread-based):

* Utilizes Web Workers to create a separate thread for profiling.
* Main thread: Runs the program evaluation.
* Worker thread: Handles profiling tasks.
* Implementation details:
  + Creates a SharedArrayBuffer for inter-thread communication.
  + Main thread updates the SharedArrayBuffer with current function information.
  + Worker thread uses setInterval to sample the SharedArrayBuffer every 1ms.
  + Samples are processed to generate a statistical profile of function execution times.
* Advantages:
  + Minimal interruption to the main program execution.
  + Consistent sampling interval, independent of program complexity.
  + Lower overhead compared to the Promise-based approach.
* Challenges:
  + Potential for slight inaccuracies due to the statistical nature of sampling.
  + Requires careful synchronization between threads to ensure data consistency.

**Evaluation:** Profilers were tested using programs with different numbers of function calls

**Results:**

1. Overhead Comparison:

Figure 1 shows the overhead of the Promise-based Statistical Profiler. As evident from the graph, this approach introduces significant overhead, making it impractical for most scenarios.

Figure1: Overhead of Promise-based Statistical Profilerグラフ, 折れ線グラフ

自動的に生成された説明

Figure 2 compares the overhead of the Event-based Profiler and the Worker Thread-based Statistical Profiler.

Figure 2: Overhead Comparison of Event-based and Worker Thread-based Statistical Profilers

グラフ, 棒グラフ

自動的に生成された説明

Key observations from Figure 2:

* Event-based Profiler: Lower overhead for <10^5 function calls, but increases linearly with the number of calls.
* Worker Thread-based Statistical Profiler: Higher but constant overhead, becoming more efficient for ≥10^5 function calls.

1. Accuracy:
   * Event-based Profiler: Higher accuracy for individual functions.
   * Statistical Profiler (Worker Thread): Good overall estimate, potentially missing very short-lived functions.
2. Scalability:
   * Worker Thread-based Statistical Profiler showed better scalability for large programs.

**Conclusion:**

* The Promise-based Statistical Profiler, proves impractical due to excessive overhead.
* For <10^5 function calls: Event-based Profiler is preferable (higher accuracy, lower overhead).
* For ≥10^5 function calls: Worker Thread-based Statistical Profiler offers better balance between accuracy and performance.
* Web Workers and SharedArrayBuffer effectively overcome JavaScript's single-threaded limitations in statistical profiling.

These findings provide valuable insights for choosing appropriate profiling strategies in JavaScript-based language implementations.

**Future Work:**

* Extend profiling techniques to other scripting languages (e.g., Python, Ruby).
* Integrate static analysis with dynamic profiling.

Related Work:

This research builds upon and extends the concepts from various existing profiling tools. Some notable related works include:

* gprof:
  + A popular profiling tool for C, C++, and Fortran programs
  + Uses a combination of statistical sampling and call graph analysis
* Valgrind:
  + A comprehensive suite of debugging and profiling tools
  + Operates at the machine code level, allowing for detailed analysis
* Python cProfile:
  + A built-in profiling module for Python
  + Offers function-level statistics including call counts and execution times