ParLOT: Efficient Whole-Program Call Tracing for HPC Applications

Saeed Taheri Sindhu Devale Ganesh Gopalakrishnan Martin Burtscher

School of Computing, University of Utah

Department of Computer Science, Texas State University

THE UNIVERSITY OF UTAH®

TEXAS STATE UNIVERSITY®

The rising STAR *of Texas*

Outline

- HPC Debugging
  - Tracing Challenges
- ParLOT Design
  - Binary Instrumentation
  - Compression Mechanism
- Evaluation
- Conclusion

HPC Debugging

- HPC bugs are expensive because:

HPC Debugging

- HPC bugs are expensive because:
  - Program failures **waste resources** (time, energy, SUs, etc.)

HPC Debugging

- HPC bugs are expensive because:
  - Program failures **waste resources** (time, energy, SUs, etc.)
  - HPC bugs are often **not reproducible**

HPC Debugging

- HPC bugs are expensive because:
  - Program failures **waste resources** (time, energy, SUs, etc.)
  - HPC bugs are often **not reproducible**
  - Information collection to detect/locate failure points introduces **overhead**

HPC Debugging

- HPC bugs are expensive because:
  - Program failures **waste resources** (time, energy, SUs, etc.)
  - HPC bugs are often **not reproducible**
  - Information collection to detect/locate failure points introduces **overhead**

In this work, we propose an **efficient whole-program tracing** infrastructure to help the HPC debugging community.

Desired Tracing Features

- "Always-on" tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- Low overhead (runtime and storage)

Desired Tracing Features

- “Always-on” tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- Low overhead (runtime and storage)

MINDSET: ***Pay a little bit more upfront to significantly reduce the number of overall debug iterations***

Desired Tracing Features

- "Always-on" tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- Low overhead (runtime and storage)

Towards automated debugging/tracing system

Desired Tracing Features

- "Always-on" tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- Low overhead (runtime and storage)

Studying behavior of application at runtime

Desired Tracing Features

- “Always-on” tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- Low overhead (runtime and storage)

Regardless of system, OS, compiler and hardware

Desired Tracing Features

- "Always-on" tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- Low overhead (runtime and storage)

Main goal of our work

Desired Tracing Features

- "Always-on" tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- Low overhead (runtime and storage)

Contribution 1: We use *Pin*, a dynamic binary instrumentation tool by Intel, to instrument binaries (regardless of source language and compiler) and capture all functions` entry/exit points including library calls for every thread/process.

Desired Tracing Features

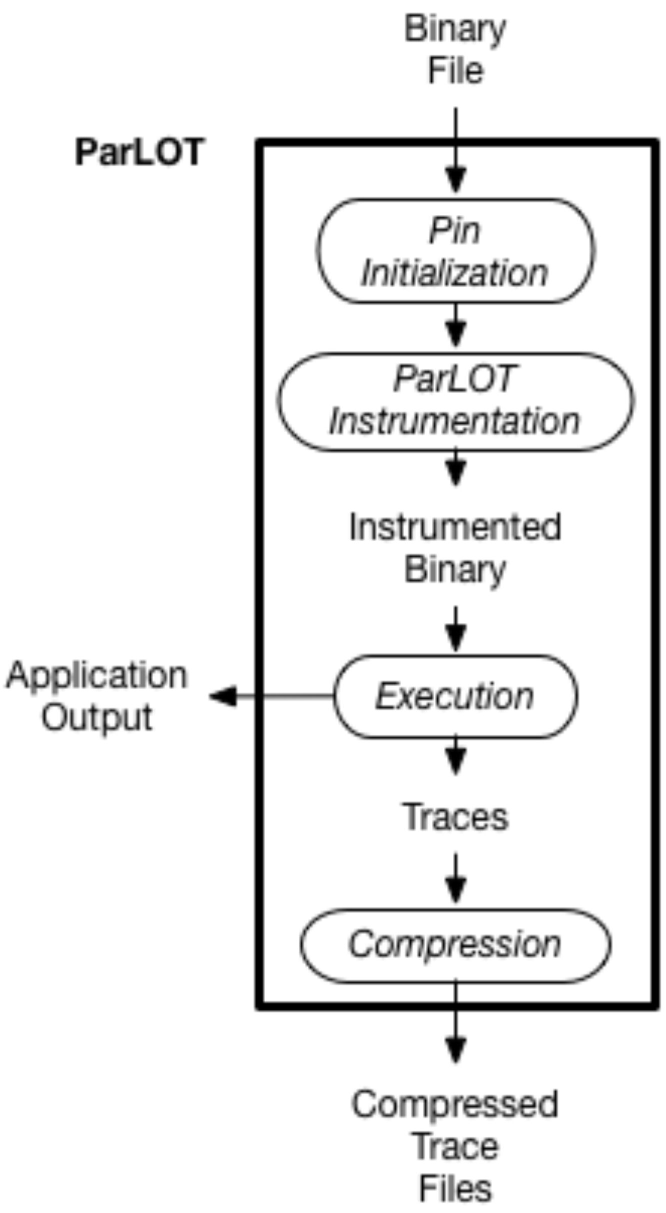
- "Always-on" tracing capability
- No source-code modification
- No recompilation
- Dynamic instrumentation
- Portability
- **Low overhead (runtime & storage)**

Contribution 1: We use *Pin*, a dynamic binary instrumentation tool by Intel, to instrument binaries (regardless of source language and compiler) and capture all functions` entry/exit points including library

Contribution 2: An incremental data compression algorithm that drastically reduces the overhead of on-the-fly whole-program tracing

ParLOT Design

- **Par**allel **L**ow **O**verhead **T**racing Tool
  - Tracing Operations
  - Incremental Compression
  - Compression Algorithm
  - Call-stack Correction



Binary Instrumentation

- ParLOT instrumentation
  - Every thread launch and termination
  - Every function entry and exit
- Separate trace file for each thread
  - Each file contains ordered sequence of function calls and returns

Binary Instrumentation

- ParLOT instrumentation
  - Every thread launch and termination
  - Every function entry and exit
- Separate trace file for each thread
  - Each file contains ordered sequence of function calls and returns
- Per-thread information
  - Thread ID
  - Current function ID
  - Current call stack
  - Current SP value (for stack correction)

Incremental Compression

- Conventional compression approaches
  - Trace first written to buffer (in memory), buffer is compressed once full
  - Threads sporadically block to compress data → highly non-uniform latency
  - Distorts trace when one thread polls data from another blocked thread
- Incremental Compression
  - Every trace element is compressed right away before writing it to memory
  - Resulting compression latency is much more uniform
  - Greatly improves fidelity of trace data

Compression Algorithm

- CRUSHER: automatic compression algorithm synthesis tool
  - Trained on traces from the Mantevo miniapps
  - Resulting best algorithm: LZ followed by ZE
  - Delivers high compression ratio with low overhead

Compression Algorithm

- CRUSHER: automatic compression algorithm synthesis tool
  - Trained on traces from the Mantevo miniapps
  - Resulting best algorithm: LZ followed by ZE
  - Delivers high compression ratio with low overhead

Call-Stack Correction

- PIN cannot identify some function exit points
  - E.g., the instructions of an inlined function may be interleaved with the caller’s instructions

Call-Stack Correction

- PIN cannot identify some function exit points
  - E.g., the instructions of an inlined function may be interleaved with the caller's instructions
- ParLOT's solution
  - Records the SP value at each function entry and exit
  - Pops the internal call stack until consistent with SP value

Call-Stack Correction

- PIN cannot identify some function exit points
  - E.g., the instructions of an inlined function may be interleaved with the caller's instructions
- ParLOT's solution
  - Records the SP value at each function entry and exit
  - Pops the internal call stack until consistent with SP value
- Other DBIs might [not] need such correction

Evaluation

- MPI-based NAS Parallel Benchmarks (input classes B and C)
- San Diego Supercomputer Center – Comet
- 1, 4, 16 and 64 compute nodes (each with 16 cores)
- Compute nodes: Xeon E5-2680 v3 processors – 28 cores – 128 GB memory
- Measured metrics
  - **Tracing overhead**
  - **Tracing bandwidth**
  - **Compression ratio**

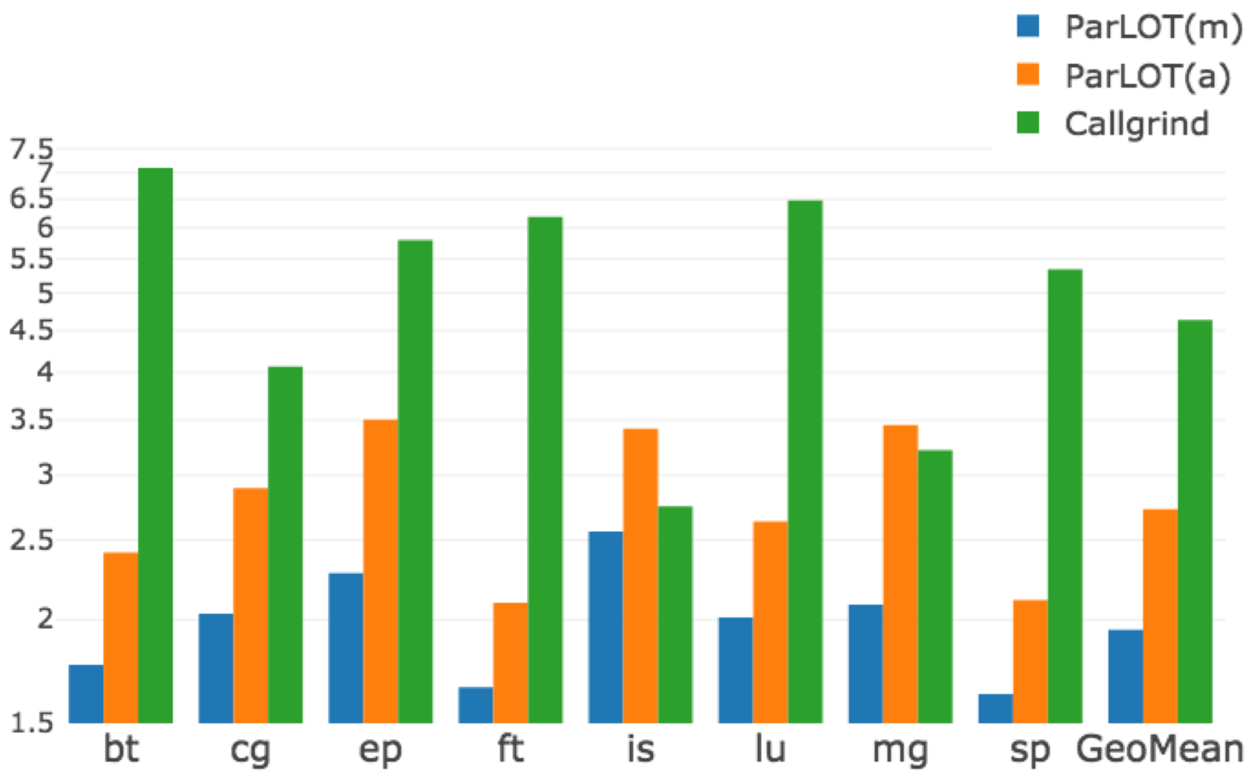
Tracing Overhead

ParLOT(m): collects traces from the main image

ParLOT(a): collects traces from all images (including library function calls)

Callgrind: DBI-based tracing tool that collects function-call graphs and performance data

Average overheads (input C)
- ParLOT(m): **1.94**
- ParLOT(a): **2.73**
- Callgrind: **4.63**



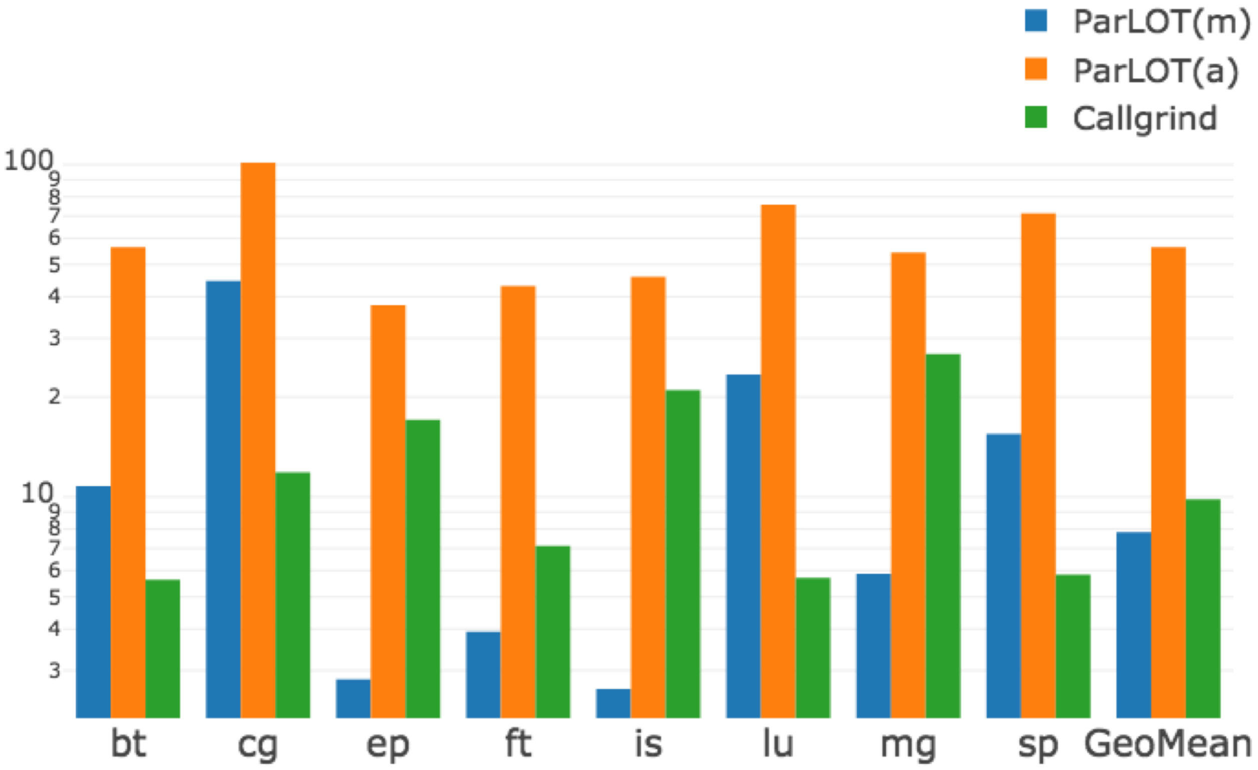
Required Bandwidth

ParLOT(m): collects traces from the main image

ParLOT(a): collects traces from all images (including library function calls)

Callgrind: DBI-based tracing tool that collects function-call graphs and performance data

Average required bandwidth on input C
- ParLOT(m): **7.8** kB/s
- ParLOT(a): **56.4** kB/s
- Callgrind: **9.8** kB/s



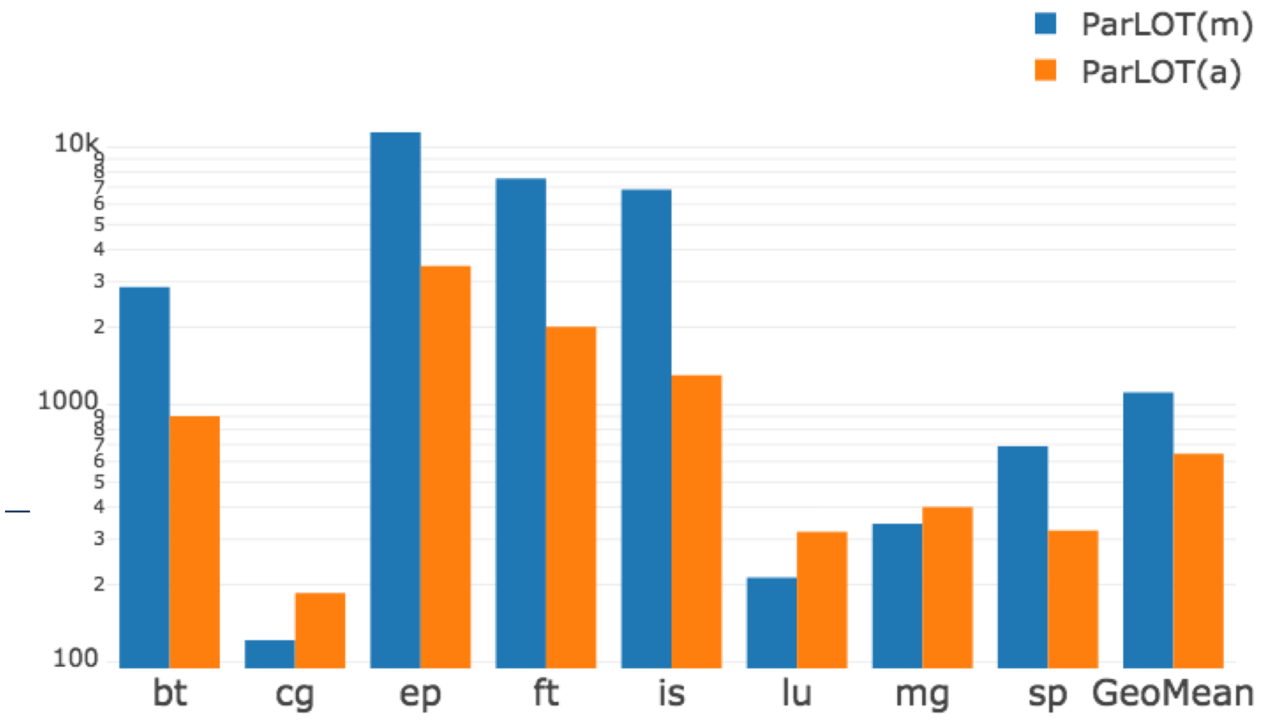
Compression Ratio

Average compression ratio of ParLOT(a) on input C: **644.3**

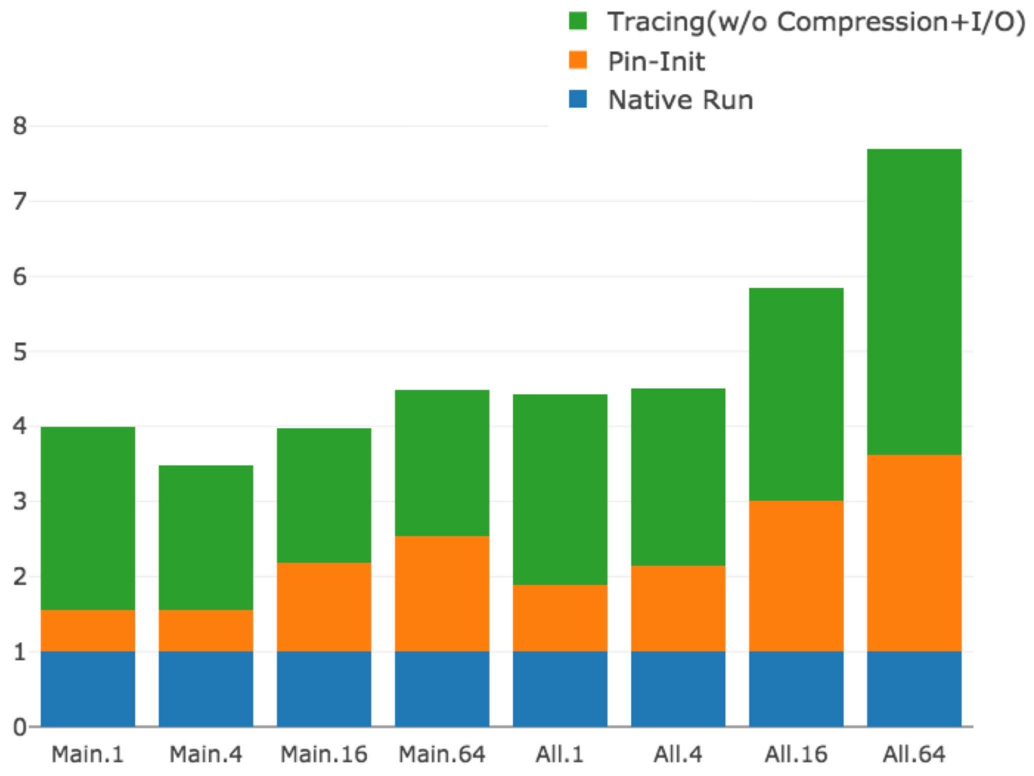
Corresponding required bandwidth: **56.4** kB/s

ParLOT can collect **36 MB worth of data** per core per second while only requiring **56 kB/s**

CG behavior: conjugate gradient method with irregular memory accesses and communication – larger number of distinct calls with more complex patterns

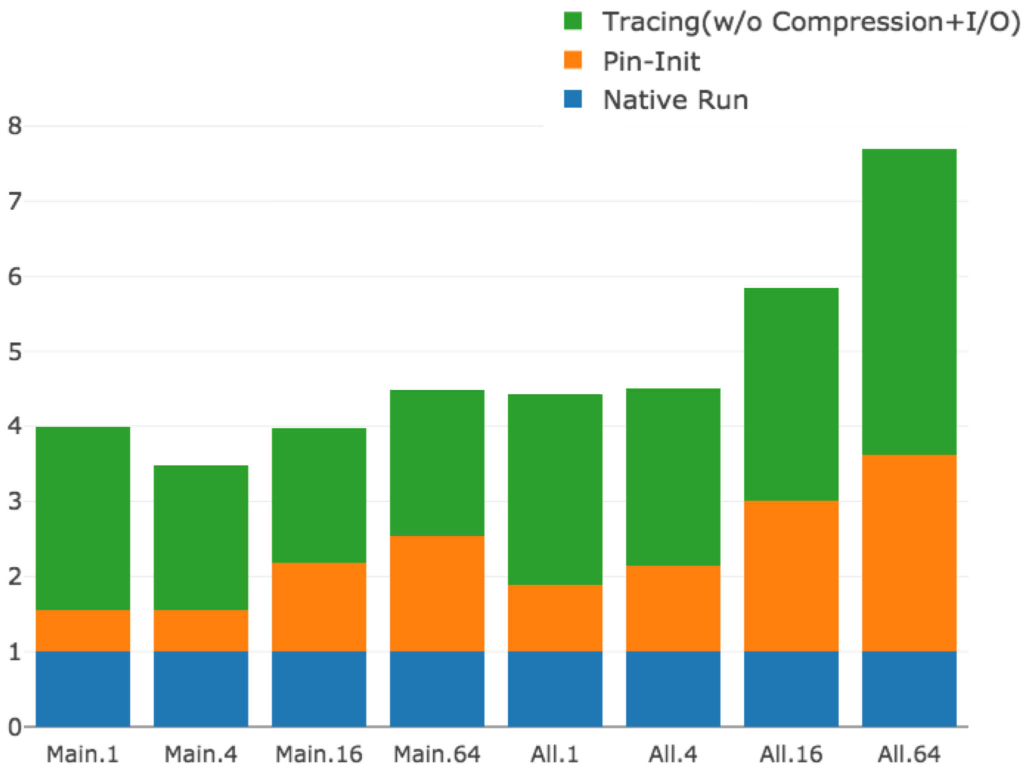


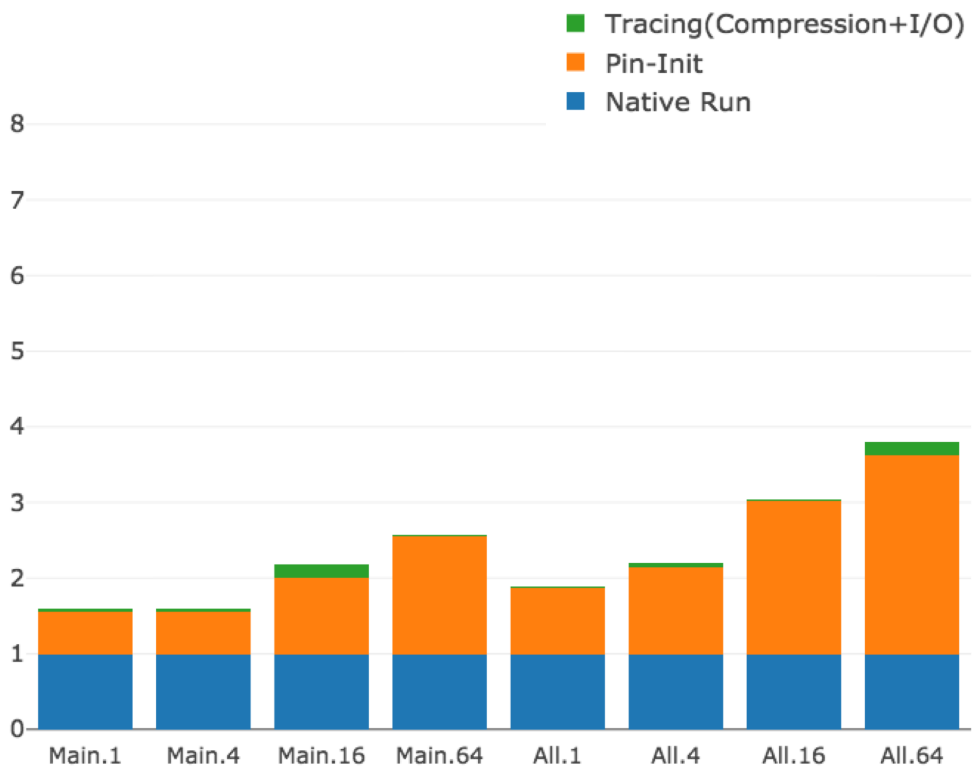
Overheads



Breakdown of average ParLOT overhead on NAS - Input C – 1, 4, 16 and 64 nodes

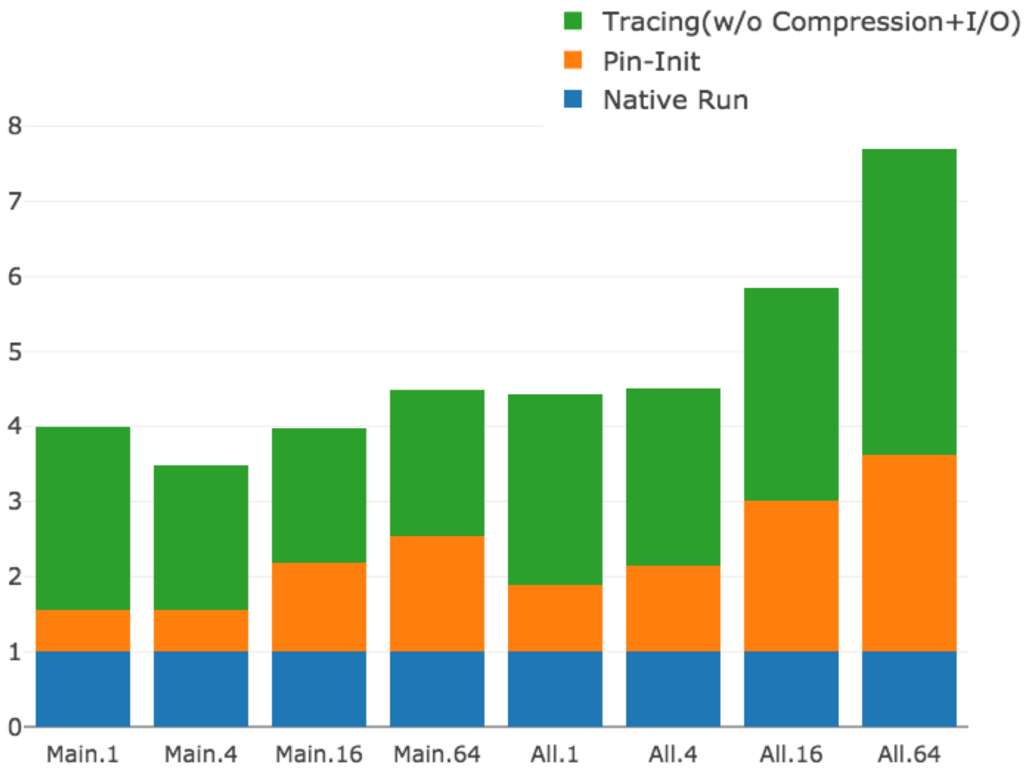
Overheads

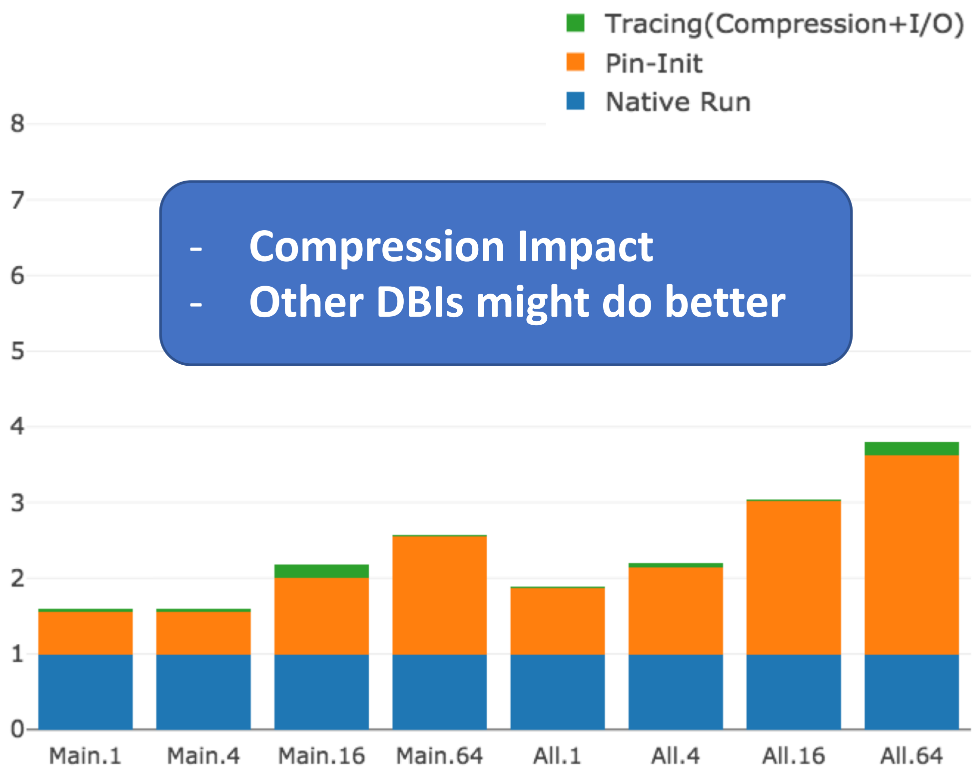




Breakdown of average ParLOT overhead on NAS - Input C – 1, 4, 16 and 64 nodes

Overheads





Breakdown of average ParLOT overhead on NAS - Input C – 1, 4, 16 and 64 nodes

Summary & Conclusion

- **ParLOT:** a portable low-overhead whole-program tracing approach that collects and compresses function-call traces on-the-fly

Summary & Conclusion

- **ParLOT:** a portable low-overhead whole-program tracing approach that collects and compresses function-call traces on-the-fly
- Includes new trace compression approach
  - Incrementally compresses trace data to make latency uniform

Summary & Conclusion

- **ParLOT:** a portable low-overhead whole-program tracing approach that collects and compresses function-call traces on-the-fly
- Includes new trace compression approach
  - Incrementally compresses trace data to make latency uniform
  - Yields high compression ratio to drastically reduce bandwidth and storage requirement

Summary & Conclusion

- **ParLOT:** a portable low-overhead whole-program tracing approach that collects and compresses function-call traces on-the-fly
- Includes new trace compression approach
  - Incrementally compresses trace data to make latency uniform
  - Yields high compression ratio to drastically reduce bandwidth and storage requirement
  - Efficiently implemented to minimize runtime overhead

Summary & Conclusion

- **ParLOT:** a portable low-overhead whole-program tracing approach that collects and compresses function-call traces on-the-fly.
- Includes new trace compression approach
  - Incrementally compresses trace data to make latency uniform
  - Yields high compression ratio to drastically reduce bandwidth and storage requirement
  - Efficiently implemented to minimize runtime overhead
- Enables comprehensive post-mortem analysis on traces (debugging, performance analysis, program understanding, etc.)

Thanks.
Any questions?