Bringing the Windows .NET Performance Diagnostics Experience to Linux

Brian Robbins

Microsoft

About Me

- Developer at Microsoft on .NET Reliability and Performance Team.
- Responsible for the performance of the entire .NET stack.
  - Runtime → Applications.

Agenda

- What is .NET Core?
- Managed vs. native applications
- How we capture performance trace data
- Demos - What works and what doesn’t
- Opportunities for future improvement

Quick Poll

Raise your hand if:

- You have done a perf investigation.
- You have done a managed perf investigation.
  - Java, Python, .NET, etc.
- Have had trouble getting stacks and symbols for managed code.
- Have tried to understand the behavior of runtime components
  - Garbage Collection, Just-In-Time Compiler, etc.

This talk is all about how we do these things for .NET on Linux.

Our Goal

- To enable use of existing Linux tracing tools to understand the behavior of .NET apps.
- If you know of something we can do to improve, we want to know about it.

.NET Core

- Write .NET apps (C#, VB, and F#) that run on Windows, Linux and OSX.
- Code is compiled down to MSIL – compiled to native at runtime.
  - Called “Managed Code”

- Standalone (can be packaged with app).
- OSS – Runtime, libraries, compiler, languages and tools.
- Supported in production by Microsoft.

- CoreCLR: Runtime responsible for app execution.
  - CLR == Common Language Runtime

Managed vs. Native Applications

- Managed applications are native applications + additional services.
  - Managed Code
  - Garbage Collection
  - Just-In-Time Compiler / Interpreter
  - Interop
- Instrumentation and special handling are needed to understand behavior.

Perfcollect - Collection Script

- What we use for Linux perf investigations.

- Installs dependencies for new users.
- Enables and disables tracers.
- “Good” defaults, modes for specific investigations.
- Produces an archive that contains everything.
  - Trace files
  - Managed/native symbols

- http://aka.ms/perfcollect

Demo: CPU Flamegraph

- Use perfcollect to collect cpu-clock events via perf.
- JIT-compiled code symbols via /tmp/perf-$pid.map.
- Use Brendan Gregg's FlameGraph tools
  - http://brendangregg.com/FlameGraphs/cpuflamegraphs.html.

.NET Grew Up on Windows

- Event Tracing for Windows (ETW)
  - High performance logger built into the OS
  - Machine-wide traces
  - CPU samples / Context switches with call stacks
  - Kernel and user-mode events with call stacks
- Symbols
  - Released software symbols published to symbol servers
  - Symbols automatically downloaded by viewer at analysis time
  - Use DLL signature (like buildid) as lookup key

- One trace file spans kernel, runtime, libraries, and app.

… And then Came to Linux

- Goal: Enable the analysis techniques from Windows using the Linux ecosystem.

- We use perf and LTTng - Similar functionality and goals to ETW.
  - Machine-wide
  - Event “driven” – Kernel and user-mode
  - Some stacks

Why Use Both Perf and LTTng?

Perf:

- Sample-based profiler with call stacks - Understand arbitrary code behavior.
- Kernel events with call stacks.
- Doesn’t support user-mode tracepoints.

LTTng:

- User-mode tracepoints – Instrument runtime services to gain further insight.
- Persistent and real-time traces – can be used for analysis and monitoring.

Missing: Stacks for usermode tracepoints.

Data Collected

Machine-Level:

- CPU samples (cpu-clock), Scheduling events (sched)

Runtime:

- Object Allocation, GC, JIT, etc.

Managed Code Instrumentation:

- App-level “tracepoints” – Ex: Start/Stop WebRequest

What We Can Do Today

Using Linux Tools:

- Resource Analysis (CPU/Blocked time)
- Managed runtime event collection
  - Anyone can write their own analysis scripts on this data.
- GC, JIT Reporting

Using Windows Tools:

- Advanced analysis of managed runtime services
  - GC, JIT, ThreadPool, Managed Exceptions, etc.
- Limited object allocation profiling – no stacks.

Internals: Stack Walking

- Compile native code with –fno-omit-frame-pointer.
- JIT and pre-compiled managed code preserve frame pointer.

- Still experience broken stacks when libraries don’t preserve the frame pointer.
  - Not a new problem.

Internals: Symbol Resolution

- CoreCLR generates /tmp/perf-$pid.map.
  - Extensibility point in perf – contains records for JIT-compiled code.
  - 00007FDA67060480 61 void [serializationtest] SerializationComparision::Main()
- perf resolves JIT compiled code using the map file.

- Sticking Point: Pre-compiled managed code.
  - Cross-platform binaries – contain IL + metadata + pre-compiled code
    - Can’t just convert to ELF – large platform specific investment.
  - Can’t use /tmp/perf-$pid.map as a workaround.
  - Generate map files that are consumable by custom .NET tools.

Internals: Symbol Acquisition

- All symbols embedded in trace archive.
- Native Symbols
  - Function names present in binary to simplify performance tracing.
  - Debug symbols must be downloaded manually.
- Managed Symbols
  - Generated at collection time.
    - JIT: Generated by runtime (/tmp/perf-$pid.map)
    - Pre-Compiled: Generated by offline utility (crossgen)
- Future: Would like a symbol server
  - Automatic consumption at analysis time of both managed and native symbols.
  - We are looking at ways to participate in this effort / considering a proposal.

Demo: GC/JIT Analysis

- Capture trace data from GC/JIT via LTTng tracepoints.
  - Instrumentation in the runtime that is emitted via LTTng.

- Run PerfView to generate reports
  - Existing .NET tool ported from Windows.
  - Can read CTF and generate reports based on trace data.

Internals: Tracepoint Generation

- Instrumentation is sent to different systems based on the platform.
  - Windows: ETW
  - Linux: LTTng

- Use build scripts to generate tracepoint definitions and stub functions that log them. Runtime instrumented with calls to stubs.

- Some runtime services are implemented in managed code. Applications can add their own instrumentation.
- Unfortunately, instrumentation in managed code funnels to one tracepoint, which makes filtering hard.
  - Want dynamic tracepoint registration.

Internals: Consuming CTF Traces in PerfView

- PerfView contains custom report generation code – consumes event stream and builds reports.
- Underlying managed trace data reader called TraceEvent
  - Not to be confused with TRACE_EVENT.
  - Reads multiple trace formats (ETW, CTF, XML, etc.)
  - Exposes standard consumption API.
  - Can handle live sessions for monitoring scenarios.
    - More to come on this in the next session from our friends at Criteo.

Wrapping Up

Opportunities for Future Improvement

Managed runtimes have extra needs over and above native code.

- Better support for JIT-compiled and pre-compiled cross-platform code (Mostly around symbols)
- Stacks for user-mode tracepoints (Tracing runtime services)
- Dynamic registration of tracepoints (Tracing from within managed code)

Our Goal

- To enable use of existing Linux tracing tools to understand the behavior of .NET apps.
- If you know of something we can do to improve, we want to know about it.

Resources

PerfCollect: https://aka.ms/perfcollect

Demo Code: https://github.com/brianrob/sample-code/tree/tracingsummit2017/talks/TracingSummit2017

CoreCLR Linux Tracing HOWTO: https://github.com/dotnet/coreclr/blob/master/Documentation/project-docs/linux-performance-tracing.md

PerfView: https://github.com/microsoft/perfview

Thank You!

Questions and Feedback:

Brian Robbins
brianrob@microsoft.com
GitHub: brianrob