JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud

Alexey Khrabrov  Marius Pirvu  Vijay Sundaresan  Eyal de Lara

University of Toronto  IBM  IBM  University of Toronto
Just-in-time compilation in the JVM

- At application build time:
  - Java/Scala/etc. source code → portable Java bytecodes

- At runtime:
  - Bytecode interpreter (slow)
    - Collects profiling data
  - Dynamic JIT compilation in the background
    - Only “hot” code paths
    - Rely on profiling data to optimize
  - Result: native code executing directly on CPU
JIT compilation performance issues

- Compiler CPU overhead
  - Up to 50% of total CPU time during start/warm-up
  - Application competes with the JIT for CPU
- Memory footprint of compiler data structures
  - Transient spikes during warm-up: up to 100s of MBs
- Have to overprovision resources
  - More CPU to maintain QoS despite JIT activity
  - Extra memory that goes unused after warm-up
- Lower application density in the cloud
Promising approach: JIT compiler *disaggregation*

- Decouple JIT from the JVM, move to separate (remote) process
  - *Software* disaggregation - similar to e.g. microservices
- Reduces memory usage
  - Spikes from multiple client JVMs unlikely to align at the server
- JIT doesn’t steal CPU cycles from application - better QoS in small containers
- Enables independent autoscaling of compilation resources
Remote JIT drawbacks

- JIT overhead only moved around
  - At the expense of communication overhead
- Can result in *higher overall CPU usage*
- Each compilation takes more CPU time
  - Networking and data serialization overheads
  - Also more wall-clock time due to latency
Our approach to improve remote JIT

- A lot of code shared between multiple JVMs in cloud workloads
  - E.g. multiple instances of the same application for autoscaling
- Can reuse JIT-compiled code in multiple JVMs
  - Cache at compiler server, send to clients running the same code
- Goal: reduce overall CPU usage by amortizing JIT cost over many JVMs
Challenges in reusing JIT-compiled code

- JIT-compiled code breaks if used as is in a different JVM
- Pointers to class metadata, other compiled methods, etc.
  - Addresses depend on the order of class loads, JIT compilations, etc.
- Assumptions: e.g. “class C currently has a single known subclass”
  - Can break even for the same application due to dynamic class loading

```
JIT-compiled X.m1()
... native code ...
```

```
<table>
<thead>
<tr>
<th>Class X</th>
<th>Class Y</th>
<th>JIT-compiled Y.m2()</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM 1:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

virtual address space

```
<table>
<thead>
<tr>
<th>Class Y</th>
<th>Class X</th>
<th>JIT-compiled Y.m2()</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVM 2:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Our solution: *serialize* JIT-compiled code

- JITServer stores cached JIT-compiled methods in *serialized* format
  - Add *serialization records*: describe how to “fix” the code in another JVM
- *Relocation records* to update addresses in compiled code
  - More difficult than e.g. relocating compiled C code
  - Cannot simply identify everything by symbol name
- *Validation records* to verify compiler assumptions
- Main building block: identifying classes *equivalent at runtime* across JVMs
  - Can express all relocations and validations in terms of Java classes
  - Same Java class definition can result in *distinct* classes at runtime
Identifying runtime classes across JVMs

- **Assign globally unique ID to each runtime class**
  - “RAMClass” in OpenJ9 - our target JVM

- **Store enough info to lookup and verify the class in any JVM**
  - Fully-qualified class name
  - Secure hashes (e.g. SHA-256) of immutable class metadata
    - “ROMClass” in OpenJ9; includes method bytecodes
    - For each class and interface in the inheritance chain
  - Class loader: identify by name of 1st loaded class
    - Heuristic that works well in practice
Serialized JIT-compiled method example

abstract class A {
    abstract void m1();
}
class B extends A {
    void m1() { ... }
}
class C {
    static void m2(A o) {
        o.m1(); // inlined as B.m1()
    }
}

Compiled method C.m2():
...
cmp rax, ramclass_B; rax contains RAMClass of o
jne slow_path
...  ; inlined body of B.m1()
.slow_path: ...  ; virtual call to o.m1()
What methods to cache

- Not a goal to cache everything
- Relocatable code can be up to ~10% slower
  - Limits possible optimizations
  - Lower peak throughput for long-running JVMs
- Particularly “hot” methods compiled with more optimizations are not cached
- JITServer cache hit rate is ~70-95% in practice
- Faster JVM start and reduced CPU usage without hurting peak performance
Performance evaluation

- **Benchmarks: 3 web applications**
  - 1. AcmeAir - airline reservation system ← *presenting this one, similar results for others*
  - 2. DayTrader - stock trading platform
  - 3. Spring PetClinic - animal hospital information system
  - (More representative of cloud workloads than e.g. Java SPEC benchmarks)

- **11 machines with 16 CPU cores each connected with 10 Gbit/s Ethernet**

- **Application instances run in Docker containers**
  - Default size: 1 CPU core, 1 GB memory (roughly AWS EC2 t2.micro instance)

- **Single JITServer instance runs on separate machine**
Application performance: start time

- Start time: from starting the JVM until ready to serve requests
- Varying container sizes
  - XS: 0.5 CPU, 512 MB; S: 1 CPU, 1 GB; M: 2 CPUs, 2GB; L: 4 CPUs, 4 GB
- Up to 58% reduction with caching, only up to 40% without
Application performance: warm-up time

- Warm-up time: from applying load until reaching 90% of peak throughput
  - Workload configured to saturate application throughput
- Up to 87% reduction with caching, only up to 80% without
Overall system efficiency: CPU cost

- Many JVM instances (up to 64 concurrently) started/stopped over 1 hour
- CPU cost: total CPU time (all JVMs + JITServer) per request served
- Up to 21% increase with remote JIT without caching
- Up to 77% reduction with caching
Overall system efficiency: memory usage

- Peak total memory usage (RSS) of all concurrent JVMs + JITServer
- Up to 62% reduction compared to local JIT
- Result: higher application density
JITServer scalability

- Warm-up time (normalized) with increasing number of clients
  - Remote JIT is effective if warm-up is faster than with local JIT
  - All clients start simultaneously; cache is initially empty
- Caching allows serving more clients using the same resources
  - Compilation cost is effectively amortized over multiple clients
Effect of network latency

- Warm-up time with increasing (simulated) network latency
  - Bandwidth has limited effect - communication pattern is latency-bound (small message pairs)
- Caching allows ~2x higher latency
  - ~54% fewer messages per compilation
  - Very good performance for typical datacenter latencies (100s of microseconds)
Current and future work

● Integration with serverless/FaaS frameworks (e.g. OpenWhisk)
  ○ Main prerequisite: automatic sizing and scaling of JITServer resources
  ○ Requires multiple relatively “small” JITServer instances
  ○ Need to share and preserve the cache to avoid JITServer “cold starts”
  ○ Using persistent snapshots to reduce synchronization and support scaling down to zero

● Prefetching cached compiled methods

● Caching and reusing profiling data
  ○ Reuse indirect JIT compilation effort for methods that are not cached (up to 30%)
  ○ Profiling data is expensive: need to interpret methods for 1000s of invocations
Summary

- JIT compilation overheads make JVM inefficient in the cloud
- Remote JIT compilation is a promising approach
  - Reduces memory usage, but increases overall CPU usage
- We make remote JIT efficient by reusing compiled code in multiple JVMs
- Novel mechanism for serializing dynamically compiled code
- Open source JITServer implementation in the Eclipse OpenJ9 JVM
  - [https://github.com/eclipse-openj9/openj9](https://github.com/eclipse-openj9/openj9)
- JITServer significantly improves performance
  - Reduces CPU and memory usage, start and warm-up time
  - Increases application density in the cloud
Questions?
Limitations of other approaches

- Static AOT (ahead-of-time) compilation, e.g. GraalVM Native Image
  - Only works for “static” subset of Java under closed world assumption
  - Lower peak throughput: lack of real profiling data; no recompilation
- Caching JIT-compiled code, e.g. SCC (shared classes cache) in OpenJ9
  - Not all code is cached, still need JIT compiler with its memory overhead
  - Pre-populated cache at build time
    - Added complexity for application developer; larger container images
  - Dynamically populated cache shared between JVMs on local host
    - Forces co-location of instances of same application on same machines
- Checkpointing or reusing “warm” JVMs
  - Similar issues: complexity; forced co-location; idle footprint
Existing work on remote JIT

- **15-20 years ago in embedded/mobile computing**
  - Main motivation: not enough resources for JIT on device
  - Overall resource usage not considered
  - All designs assume that each remote compilation is single request-reply
    - All information used by compiler known before sending request
    - Not feasible in a modern JVM with a complex JIT

- **Recent (Dec 2021): Azul Cloud Native Compiler**
  - Proprietary; very limited design information
  - Seems to focus on giving the JIT more CPU and memory, not on overall resource usage
Other challenges in remote JIT compilation

- Need to request JVM runtime info on demand
  - Hard to determine in advance what exactly compiler will need
- Aggressive caching to reduce number of messages
- Class metadata and class hierarchy information
  - Need to invalidate caches when necessary to ensure correctness
  - Careful synchronization with class loading/unloading/redefinition
- Profiling data
  - Cache slightly outdated info to reduce amount of communication
  - Can tolerate imprecision without affecting correctness
JITServer reliability and security

- Better reliability than other types of disaggregation
  - No shared hard state
  - Client can switch to another JITServer instance or local JIT
  - Compiler crashes don’t bring down the whole JVM

- Security model
  - JITServer and all clients are in the same security domain (trust each other)
  - Encrypted communication (adds ~5% overhead)
Class loader identification across JVMs

- Runtime classes in JVM exist in the context of their **class loaders**
  - Same class loaded by different class loaders results in multiple *distinct* runtime classes
  - Need class loader instance to lookup class by name at runtime
  - Need to associate each class with its class loaders when serializing JIT-compiled code

- **Challenge:** class loaders are Java heap objects
  - Do not persist outside of running JVM process

- **Heuristic:** identify class loaders by *name of 1st loaded class*
  - Works well in practice: no failures in any applications we tried with JITServer
  - Failures in edge cases can only affect performance
    - E.g. code takes slow path
  - Correctness is always guaranteed
Compilation request latencies

- Short compilations take longer remotely than locally due to network latency
- Long compilations are only faster at JITServer if it has lots of CPU resources
- JITServer cache hits are faster than most local compilations