Preparing web applications for Loom

Mark Thomas
Staff Engineer
May 2023
Introductions

Mark Thomas - markt@apache.org
Tomcat since 2003
Committer, PMC member

Commons (Daemon, Pool, DBCP, BCEL)
Committer, PMC member

ASF member, ASF security team, ASF infrastructure team, Director 2016 to 2019
VP, Brand Management since 2018

Java EE Expert groups for Servlet, WebSocket, Expression Language
Jakarta Servlet, Pages, WebSocket and Expression Language
Committer

Staff Engineer at VMware
“Project Loom is intended to explore, incubate and deliver Java VM features and APIs built on top of them for the purpose of supporting easy-to-use, high-throughput lightweight concurrency and new programming models on the Java platform.”

https://wiki.openjdk.org/display/loom
A Brief History of Servlet Scalability
A Brief History of Servlet Scalability

HTTP/1.0
HTTP/1.1 and keep-alive
Tomcat, blocking I/O (BIO) and thread starvation
Tomcat, non-blocking I/O (NIO / NIO2)
Servlet asynchronous API and non-blocking I/O
A Brief History of Servlet Scalability

Key

Connection

Request

Web Application

Service

Tomcat & JVM

OS Scheduler
A Brief History of Servlet Scalability

HTTP/1.0

Connect, make request, close
One thread per connection
Maximum connections ==
Maximum concurrent requests ==
Thread pool size

Creating connections is (relatively) expensive
A Brief History of Servlet Scalability

HTTP/1.1 keep-alive

HTTP/1.0 had keep-alive with issues with interoperability
HTTP/1.1 fixed the issues
Better (lower) latency
Worse scalability
Typically uses more threads than there are concurrent requests
Thread starvation
Tomcat BIO connector disabled
HTTP keep-alive for the last 25% of threads in the thread pool
A Brief History of Servlet Scalability
Non-blocking I/O part 1 – between requests

Tomcat NIO / NIO2 connectors
Use non-blocking I/O while waiting for a new request
Only use a thread for connections where there is a request to be processed
Maximum connections
Maximum concurrent requests
Thread pool size
HTTP keep-alive latency benefits
Improved scalability
Use non-blocking I/O to communicate with services

Only use a thread for connections where there is a request actively being processed

Maximum connections

Maximum concurrent requests

> Thread pool size

Further improved scalability
Virtual Threads
Virtual Threads

Pre-Java 21 threads referred to as platform threads

Virtual threads

• Not linked to OS thread
• Use the heap for stack
• Have their own scheduler
• Created for a task and then allowed to terminate
• Do not pool virtual threads

Virtual thread scheduler has a pool of platform threads to do the work

• One platform thread per processor by default
Virtual Threads
Blocking operations

Platform threads
• Thread waits for operation to complete

Virtual threads
• Non-blocking operation started
• Virtual thread suspended and platform thread released
• Operations completes
• Virtual thread resumed and becomes eligible to be scheduled
• Execution continues

Virtual threads are effectively non-blocking for many blocking operations
• Increased scalability for “free”
Virtual Threads
Coding constraints

ThreadLocals
- Providing context across an API boundary OK
- Caching could be problematic

Beware of pinning
- Long lasting synchronized blocks are problematic
- Brief synchronized blocks are fine
A Brief History of Servlet Scalability

Virtual threads

Impact on throughput?
Impact on scalability?
Impact on GC?
Impact on memory footprint?
Impact of extra scheduler?
Impact on code complexity?
Impact of constraints?
Investigations

Lots of areas to explore
Areas are not independent
Try and focus on a single variable
Performance tests only ever indicative
Not meant to be representative of real applications
Java 21 is still in Early Access
This work is just a starting point
Throughput
Throughput

Aims:

• compare virtual and platform threads in same scenario
• minimise impact of other factors
• not looking to identify maximums
• relative, rather than absolute, results were primary interest

Examined:

• Different sized requests
• Different concurrencies
• Configured to minimise Tomcat and web application processing time
• Details at https://spring.io/blog/2023/02/27/web-applications-and-project-loom
Throughput Results

The bigger the response size, the less the difference

Platform thread performance is worse with concurrency of 2 than it is with 1

Virtual threads have higher throughput and this is more obvious with smaller response sizes

Once concurrency exceeds processor count, virtual threads show increased throughput compared to platform threads

Tomcat’s thread pool uses LinkedBlockingQueue for the task queue by default.

The virtual thread scheduler uses a work stealing queue by default.
Throughput
Draft results

Concurrent requests >> processors / cores available

Work in progress

• Need to ensure results are repeatable
• Are factors other than high concurrency affecting the results?
• Are results affected by number / proportion of blocking calls in application?

Draft results indicate

• Virtual threads have higher throughput
• Virtual threads have wider spread of latency

Consistent with different queuing implementations used
Easy to Use
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    resp.setContentType("text/plain");
    resp.setCharacterEncoding("UTF-8");
    ServletInputStream sis = req.getInputStream();
    byte[] buffer = new byte[8192];
    int read = -1;
    int totalBytesRead = 0;
    while ((read = sis.read(buffer)) > -1) {
        if (read > 0) {
            totalBytesRead += read;
        }
    }
    ServletOutputStream sos = resp.getOutputStream();
    String msg = "Total bytes written = [" + totalBytesRead + "]";
    sos.write(msg.getBytes(StandardCharsets.UTF_8));
}
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    resp.setContentType("text/plain");
    resp.setCharacterEncoding("UTF-8");
    AsyncContext ac = req.startAsync();
    CounterListener listener =
        new CounterListener(ac, req.getInputStream(), resp.getOutputStream());
}

private static class CounterListener implements ReadListener, WriteListener {
    private final AsyncContext ac;
    private final ServletInputStream sis;
    private final ServletOutputStream sos;
    private volatile boolean readFinished = false;
    private volatile long totalBytesRead = 0;
    private byte[] buffer = new byte[8192];

    private CounterListener(AsyncContext ac, ServletInputStream sis, ServletOutputStream sos) {
        this.ac = ac;
        this.sis = sis;
        this.sos = sos;
        sis.setReadListener(this);
        sos.setWriteListener(this);
    }

    public void onDataAvailable() throws IOException {
        int read = 0;
        while (sis.isReady() && read > -1) {
            read = sis.read(buffer);
            if (read > 0) {
                totalBytesRead += read;
            }
        }
    }

    public void onAllDataRead() throws IOException {
        readFinished = true;
        if (sos.isReady()) {
            onWritePossible();
        }
    }

    public void onWritePossible() throws IOException {
        if (readFinished) {
            String msg = "Total bytes written = [" + totalBytesRead + "]";
            sos.write(msg.getBytes(StandardCharsets.UTF_8));
            ac.complete();
        }
    }

    public void onError(Throwable throwable) {
        ac.complete();
    }
}
Easy to Use

Aims:
• Compare virtual threads with blocking code to thread pool with non-blocking
• Minimise other factors

Examined
• External service that blocked and waited a preset time before continuing
• Service ‘delay’ dominated initial results
Easy to Use

Virtual threads generally a little more performant

Difference more noticeable at low concurrency and when concurrency exceeds processor cores

Performance of blocking code with virtual threads is comparable to refactoring to use non-blocking APIs
Conclusions
Conclusions

Applications currently using non-blocking APIs will likely see minimal differences with virtual threads

Applications currently using blocking APIs
  • will likely see minimal throughput differences with virtual threads
  • will likely see measurable scalability improvements with virtual threads

Code changes may be required for:
  • long lasting synchronized blocks
  • ThreadLocals
Next Steps
Next Steps

Tomcat will include virtual thread support in June releases

Tomcat 11
  • will require a minimum Java 21

Tomcat 8.5, 9.0 & 10.1
  • No change to minimum Java versions
  • Will require Java 21 to use virtual threads

Future Tomcat development
  • Use of ThreadLocal vs. SynchronizedStack vs. new Object()
  • Investigate bottlenecks as they get reported
Thank You