Heuristic Build Tool Comparison

Dylan Bray (db37299)
University of Texas at Austin
Electrical and Computer Engineering

Alex Issa (api236)
University of Texas at Austin
Electrical and Computer Engineering

ABSTRACT

Build tools underpin nearly every aspect of software development. However, these tools often waste time and resources by performing redundant work when changes are small and incremental. In this paper, we introduce a performance analyzer that is used to collect data on build executions, then we evaluate three build tools - Maven, Gradle, and Bazel - with this data. We discuss the effects of certain types of code changes, along with other operational conditions, on each tool evaluated on a representative set of Java programs. These programs include a basic calculator application we designed, Apache Commons Math, and Google Guava. We then conclude with a discussion of the trade-offs inherent in using each tool and other lessons learned about each tool while performing the evaluation.

1. INTRODUCTION

Build tools are an important aspect of the software development ecosystem. Unfortunately, tools often rebuild code redundantly, wasting time, computational resources, and memory. We evaluate different build tools on representative Java codebases, comparing their performance handling different types of changes to the codebase. This evaluation considers Maven, Gradle, and Bazel - a new, high-performance tool that claims to handle incremental builds especially well. It compares each build tool's speed, CPU usage, and memory usage. All of these tools run on Java, lending themselves to be easily compared.

Although there are few automated tools available to convert Maven or Gradle to Bazel, there are many guides that take advantage of a nearly one-to-one mapping between the build languages [1]–[3]. We used these guides to help us perform our conversions. Our analysis includes several Java projects, including a simple, basic Java calculator application, Apache Commons Math [4], and Google Guava [5]. The basic calculator application served as our baseline and was first built using Maven, then built with Gradle, and finally built with Bazel, and served as validation that we could build a project with all three tools. We then built Apache Commons Math with all tools, followed by Guava. We also generated dependency graphs that helped us to understand the way each build tool views the organization of each codebase [6], [7].

After all projects were successfully built with every build tool, we designed the Build Tool Performance Analyzer (BTPA). The BTPA is a PowerShell script that launches a build tool and a Python script that tracks the CPU usage, memory usage, and time to completion of each build execution. We then rebuilt each project from scratch and collected performance metrics. Finally, we explored making changes to the code and collected performance metrics on each incremental build (from the original), allowing us to make comparisons between build tools across different types of changes to the code.

2. BACKGROUND: BUILD TOOLS

Apache Maven was first released in 2004 with the idea of conceptually building on top of tools such as Apache Ant and Make. Maven focuses on using predefined conventions for how to build a given project, meaning the user only needs to specify what to build and what the dependencies are, not how to build it. This is a stark difference from its main predecessors, which rely on explicit user specifications to build the project [8]. Technically, Maven does support incremental builds, but it is not well supported by Apache, though there are plugins from the community that try to enable this [9], [10].

Gradle was first released in 2007 and extends concepts of Apache Ant and Apache Maven but introduces a domain-specific language in contrast to those two previous tools. It was designed for large, multi-project builds and operates based on a series of tasks that can run in parallel [11]. It supports incremental builds by building a dependency graph and only re-executing tasks on parts of the project that are out of date or depend on parts that are out of date. Gradle also runs a daemon process that keeps data hot in memory, providing especially fast incremental builds [12].

Bazel was first released in 2015 with the goal of speeding up build processes by allowing target inputs and outputs to be fully specified and thus completely known by the system, allowing for improved performance. This is due to more accurate determination of outdated artifacts in its dependency graph [13]. It also allows Bazel to operate deterministically, so that a build on a developer laptop executes in the same way that a build on a Continuous Integration (CI) system executes [14].

1. Download a given project
2. Configure for Maven, Gradle, and Bazel
3. Test that the build is working correctly
4. Collect baseline metrics
5. Outline changes to test on

Figure 1: Basic Project Workflow Setup

```
<plugin>
  <groupId>com.github.ferstl</groupId>
  <artifactId>depgraph-maven-plugin</artifactId>
  <version>3.3.0</version>
  <configuration>
    <graphFormat>dot</graphFormat>
    <createImage>true</createImage>
    <showDuplicates>true</showDuplicates>
  </configuration>
</plugin>
```

Figure 2: Maven Dependency Graph Configuration

3. EXPERIMENTAL SETUP

We envisioned the basic workflow shown in Figure 1 to systematically set up our codebases for analysis, and we use this section to further explain this workflow. Additionally, we explain steps 4 and 5 in more detail about how we collected metrics for a given build tool and project in the next section. In order to analyze the performance of Maven, Gradle, and Bazel, we selected three Java projects: a simple calculator app with a module for each arithmetic function, Apache Commons Math version 3.6.1, and Google Guava version 30.0. We built each of these packages with Maven version 3.6.3, Gradle version 6.6.1, and Bazel version 3.7.0. We chose to use the most recent stable release of each Java project and each build tool, as of the beginning this project.

We built the calculator package to emulate a simple project that still contains a meaningful dependency structure. The main app consists of a `Calculator` class that depends on `Add`, `Subtract`, `Multiply`, and `Divide` classes that perform the main calculations. In addition, `Multiply` and `Divide` both depend on `Add` and `Subtract`.

After choosing these packages, we obtained all other source code from GitHub. We cloned each repository and built each with Maven first. We then used the Gradle conversion tool [1] to run `gradle init` which parsed `build.xml` to generate a `build.gradle` file. Overall, the generated file tends to work pretty well. Of the three packages we tested, only Guava required some minor adjustments, as their source directory structure did not match the traditional `src/main/java` structure, and there was also a rogue line that said `system 'jdk:srczip:999'` in `guava-30.0/guava/build.gradle` and needed to be removed [15]. There was also an issue with Javadoc generation, and since this fell outside of the scope of our project, it was necessary to remove `withJavadocJar()` from `guava-30.0/guava/build.gradle`.

Bazel, on the other hand, proved to be very difficult to use. The most systematic way to produce correct Bazel `BUILD` and `WORKSPACE` files involved looking at Maven dependencies and converting them to Bazel's format. Once this was done, we attempted to run `bazel build //:target`, which would fail, but thankfully also gave useful information about which dependencies or other errors needed to be fixed. This proved to be very tedious, and we also learned that Bazel does not support using local machine Maven repositories. To get around this, we simply hosted our local Maven repository with a Python server by running `python -m http.server <port>` and added `http://localhost:<port>` as a repository in the Bazel project. Furthermore, in order to generate a functioning `.jar` that correctly captures necessary dependencies in the package, you must run `bazel build //:target_deploy.jar`, which was not well documented and a source of much frustration and consternation.

After successfully building the projects and testing that the generated `.jars` worked, we moved on to generating dependency graphs, as this would enable us to have some insight on potential project modifications we could make. Doing this with Maven was relatively easy; we simply added the `depgraph-maven-plugin` version 3.3.0 [6] to the build `<plugins>` section of each `pom.xml`, as shown in Figure 2, and ran `mvn depgraph:graph` to generate a dependency graph `.png` image.

For Gradle, adding `gradle-dependency-graph-generator-plugin` version 0.5.0 [7] to every `build.gradle` file in the package, then running `gradle generateDependencyGraph` was all we needed to do. Last, Bazel has a built-in dependency graph generator you can use by simply running `bazel query 'deps(//:<PROJECT_NAME>)' --notool_deps --noimplicit_deps --output graph > graph.in`. From there, switch the `graph.in` file to use ANSI encoding, and finally run `dot -Tpng graph.in -o graph.png`. With all of these pre-requisites set up, we moved forward to focusing on how we could collect metrics on these build tool executions.

Note that the entire experiment was performed under relatively strict constraints. Every build was run on a home laptop with 32GB memory and a 6-core Intel i7 CPU running at 2.6 GHz. Each codebase was also relatively small. In addition, every tool runs on Java, and we were unable to easily isolate every Java process to track the build tool's performance. Therefore, it was necessary to prevent other Java programs from running while collecting data for the experiment. These factors may influence or limit the overall applicability of our results; however, we chose to focus on interesting areas that we believe would hold across environments.

4. PERFORMANCE MONITORING

The last step in this process involved running each build tool and monitoring its CPU usage, memory usage, and time of execution. Unfortunately, we were unable to find a good, lightweight tool to accomplish this. We needed a tool to launch the build process, begin monitoring its performance, and then log and graph the results, a workflow we outline in Figure 3. We introduce the Build Tool Performance Analyzer (BTPA) to fill this gap. It consists of a Windows PowerShell script that accepts certain arguments, finds the running Java daemon processes, and launches a Python script that monitors these processes and all of their children.

Figure 4 shows the pseudo-code for our PowerShell script. The inputs of the script are the name of the `tool` to run, the `path` to the build root, the `name` the user wants to use

1. Clean project
2. Revert changes to baseline
3. Build baseline project
4. Make code changes based on list of things to test
5. Run performance monitoring script
6. Collect resulting graphs from `/results` folder
7. Repeat until no more tests

Figure 3: Basic Script Workflow

```
input
  tool
  path
  name
  iteration
  flags
    -clean
    -cold
    -test

for each iter in iteration
  if(clean)
  then
    delete existing build files
  endif
  if(cold)
  then
    shut down build tool process
  endif
  if(test)
  then
    include appropriate tests flag
  endif
  Run tool on project in location path
  Launch monitoring process
  Save performance logs and graphs
endfor

Generate and save dependency graph
```

Figure 4: BTPA Shell Launcher

to label their test, the number of iterations to run the tool named `iteration`, and the flags to decide on doing a `-clean` start, `-cold` start, and a run with `-tests`. The PowerShell script takes these inputs and flags and uses them to determine what commands to run for the given build tool. After potentially cleaning the build, killing any Daemon processes, and adjusting the build command for testing flags, it runs the build as a Daemon process, then passes the Java process PID(s) to the monitoring Python script. These Java PIDs were acquired by using a combination of Unix `ps`, `grep`, and cleaning commands. After each build terminates, the Python script finishes the log file and performance graphs, then after all iterations are done, the dependency graph of the build is generated. These results are what we use to evaluate the given Build Tool, and the parameters such as to use tests or not, cold start or not, and clean the build directory or not, are significant ways we can find new insights in addition to modifying the codebases themselves.

Figure 5 shows the pseudo-code for the Python monitoring script, which we re-wrote from code we found on GitHub [16]. The script takes `parent_procs` as an array of running Java process IDs (PIDs), where `parent_procs[0]` is the main process that starts the build tool. From there, each PID is converted into a Process object, and added to the array variable called `watched_procs`. The `main_proc` variable is the main process object that started the build tool, which is the first process to start up, and the last process to die, aside from the Gradle/Bazel Daemon process that always runs for hot starts. The `log` variable is a map of metrics (CPU, RMem, VMem, Time) to arrays of floats, aka values at the given timestamp. Finally, the `time` variable is the current timestamp, recorded approximately every 0.1 seconds.

The basic idea of the Python script in Figure 5 is so long as the initial process is running, check for new Java processes and children of those Java processes, add them to the `watched_procs`, remove the dead processes, then tally up the total CPU, Virtual Memory, and Real Memory used by all of these processes. From there, the Python script logs these recordings at the given timestamps, sleeps for 0.1seconds, and then checks to start again, repeating this until the `main_proc` is dead. Finally, it uses the data stored to create the CPU, Virtual Memory, and Real Memory performance graph with respect to time. In terms of Python source code used, this code relies heavily on the `psutil` package [17], as well as some built-in Python packages and the `matplotlib` package for graphing. The reason the script samples every 0.1 seconds is that this is the finest granularity the `psutil` package can accomplish, while still guaranteeing accuracy.

5. RESULTS

We began our evaluation by building each package with every build tool as a baseline, and also compared performance between cold and hot starts with Gradle and Bazel, since they spawn daemon processes that run in the background to try and make future builds faster. We included a subset of our results graphs in this paper; more can be found in the project Git repository. As an example, consider the graphs in Figures 6, 7, 8, 9, and 10, where Gradle and Bazel easily beat the performance of Maven, which was expected. However, somewhat surprisingly, Gradle beat the performance of Bazel consistently. This may be due to limitations of our evaluation in that our codebases are all relatively small and we are running on a laptop, not an enterprise system. Bazel also requires

```
input
  parent_procs

var
  watched_procs
  main_proc
  log
  time

main_proc = Process(parent_procs[0])
for each PID in parent_procs
  add Process(PID) to watched_procs

time = 0
while main_proc is Alive do
  var current_cpu = 0
  var current_real_mem = 0
  var current_virtual_mem = 0

  for each proc in watched_procs
    for each child in get_all_children(proc)
      if(child is NOT in watched_procs)
      then
        add child to watched_procs
      endif
    endfor
  endfor

  for each proc in watched_procs
    if(proc is NOT Alive)
    then
      remove proc from watched_procs
    else
      add proc.CPUConsumed to current_cpu
      add proc.RealMemoryConsumed to
    current_real_mem
      add proc.VirtualMemoryConsumed to
    current_virtual_mem
    endif
  endfor

  add current_cpu to log's CPU array
  add current_real_mem to log's RMem array
  add current_virtual_mem to log's VMem
    array
  add time to log's Time array

  Use log Map to create/append to log file
  Sleep with duration of 0.1 seconds
  time = time + 0.1
endwhile

Use log Map to create CPU/Memory performance
     graph over time
```

Figure 5: BTPA Python Monitor

detailed manual configuration for performance while Gradle works "out of the box." It is possible we do not have the most optimal configuration for our projects. Another interesting result we noticed during our first try collecting data was that hot starts do not appear to experience any better performance than cold starts in our experimental setup. We do not have any hypotheses as to why this is the case.

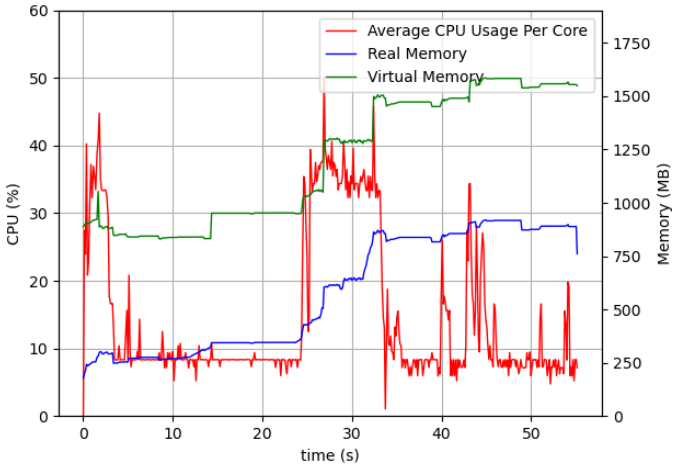

Figure 6: Maven Building Guava

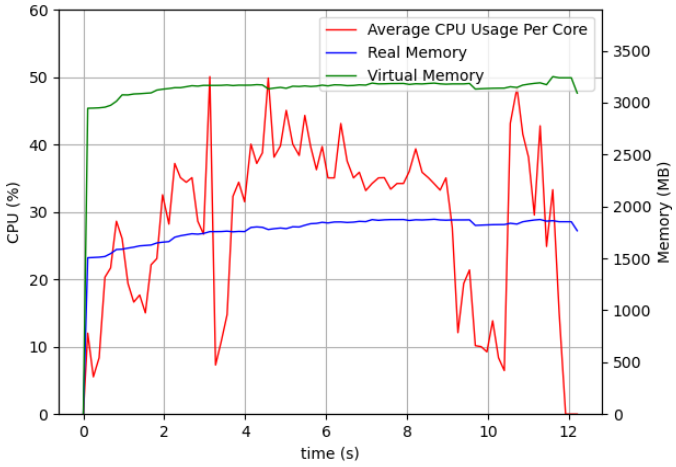

Figure 7: Gradle Cold Start Building Guava

We next made a single change at a specific location in the codebase and built each project. Building Commons Math provided some interesting insight into the behavior of each tool: Gradle appeared able to detect the incremental change while Bazel appeared to rebuild everything. We can see this conclusion by observing the time to completion of Figures 11, 12, 13, and 14. This contrasted starkly with our initial thoughts, where we assumed Bazel would get better performance in this circumstance. However, we spent much time learning about Bazel and attempting to configure each project and may not have configured the projects in the most optimal

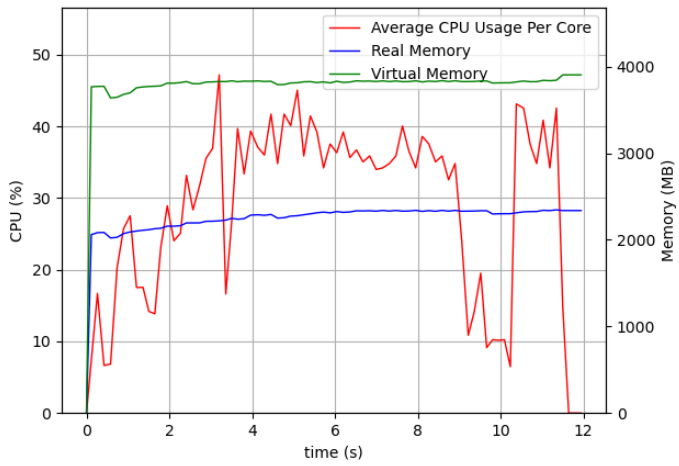

Figure 8: Gradle Hot Start Building Guava

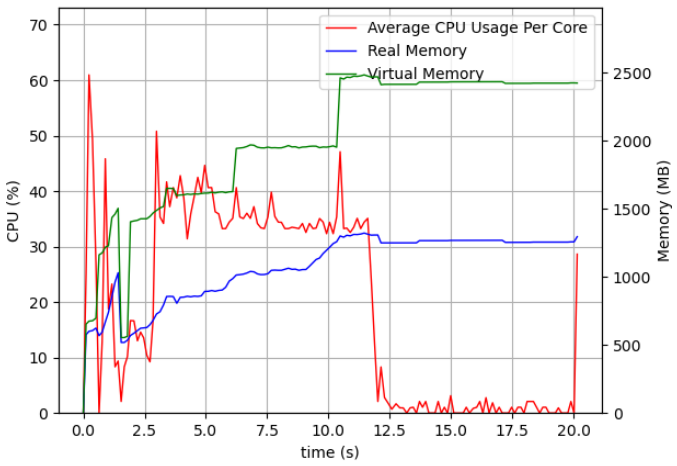

Figure 10: Bazel Hot Start Building Guava

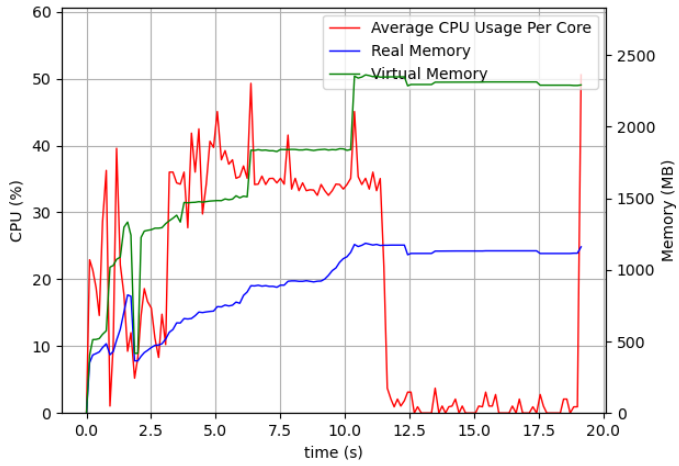

Figure 9: Bazel Cold Start Building Guava

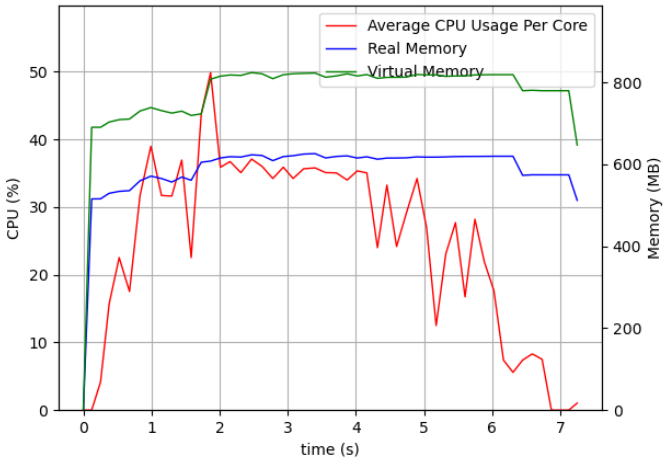

Figure 11: Gradle Clean Building Commons Math

way. This is due to Bazel requiring the user to specify exacting details about the build which, although it may increase performance when done completely correctly, places a hefty burden on the programmer. We omit discussion of Maven in this section because it does not support incremental builds.

We then made changes to every codebase in every separate project module. As shown by Figures 15, 16, and 17, Gradle performs admirably at detecting changes and only building the required elements. Figure 15 shows our simple calculator app built from scratch, and Figure 16 shows it building again with no changes. Figure 17 then shows Gradle building calculator with changes in every function sub-module, as well as the parent class. This took less time to build than the original build because all other components of the package were already loaded into the Gradle dependency folders. These results do a good job showcasing Gradle's dependency graph generation capabilities and incremental build support. We also see improved performance with Bazel on this same application. Its performance is so fast that our data collection tool is unfortunately unable to collect enough samples during execution to analyze it. It is very fast on original compilation and it is incredibly fast during an incremental update.

We also compared build executions that ran test suites. As expected, Gradle was able to run tests faster than Maven due to its parallelization capabilities, however, we were unable to migrate all tests to Bazel to extend this comparison.

6. CONCLUSION

Our analysis of the build tools we evaluated - Maven, Gradle, and Bazel - concludes that while Bazel is great leap forward and can sometimes unleash great performance, it is often not worth the trouble. Gradle, on the other hand, far exceeds the capabilities and performance of Maven, and is almost as well supported. For most tasks, a Gradle project is convenient to set up, performant, and well supported. Maven's strength lies in its longstanding status, existing integrations with programming tools, and its large, active com-

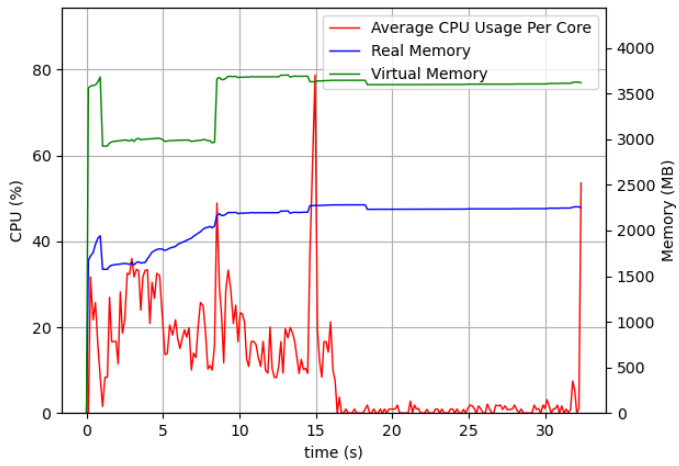

Figure 12: Bazel Clean Building Commons Math

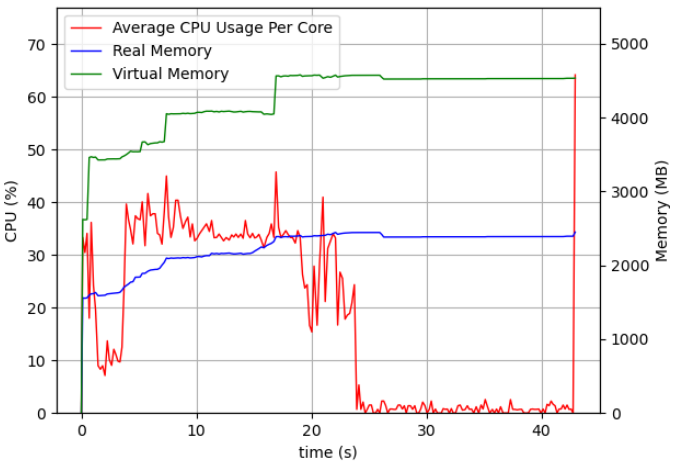

Figure 14: Bazel Incremental Building Commons Math

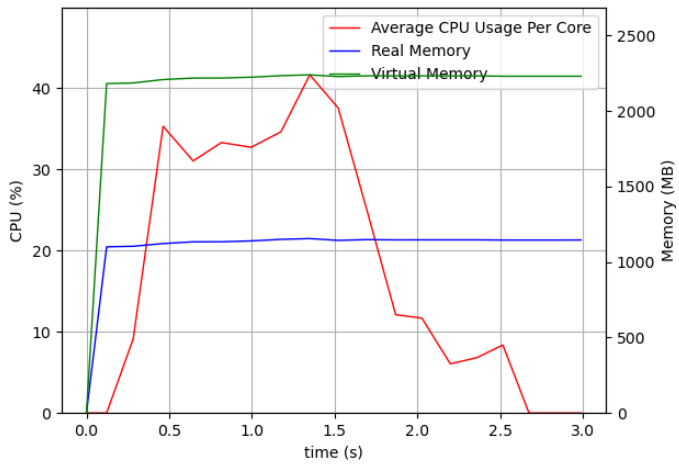

Figure 13: Gradle Incremental Building Commons Math

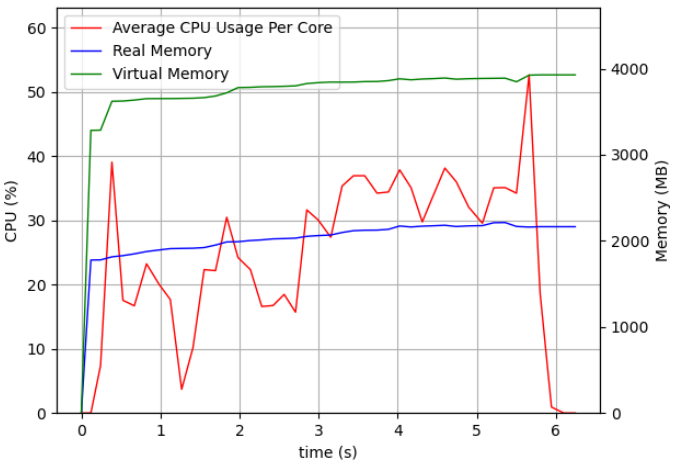

Figure 15: Gradle Clean Building Calculator

munity, but these advantages are outweighed by those of Gradle.

Perhaps one of the more interesting results of our work is the development of a flexible tool that launches a project build, collects a log of performance data on its execution, and outputs performance graphs for analysis. It became clear after a few weeks of working on this project that we bit off a bit more than we could chew and needed to focus our work. Learning how to use and truly understanding these build tools was taxing and time consuming. However, it is our hope to continue familiarizing ourselves with these tools and eventually using our tool to validate Bazel's claims of performance.

References

[1] (Oct. 2020). "Migrating builds from apache maven," [Online]. Available: `https://docs.gradle.org/current/userguide/migrating_from_maven.html#migmvn:automatic_conversion`.

[2] (Oct. 2020). "Migrating from maven to bazel," [Online]. Available: `https://docs.bazel.build/versions/master/migrate-maven.html`.

[3] N. Silnitsky. (Oct. 2020). "Migrating to bazel from maven or gradle? part 1 — how to choose the right build unit granularity," [Online]. Available: `https://medium.com/wix-engineering/migrating-to-bazel-from-maven-or-gradle-part-1-how-to-choose-the-right-build-unit-granularity-a58a8142c549`.

[4] (Aug. 2016). "Commons math: The apache commons mathematics library," [Online]. Available: `https://github.com/apache/commons-math`.

[5] (Oct. 2020). "Guava: Google core libraries for java," [Online]. Available: `https://github.com/google/guava`.

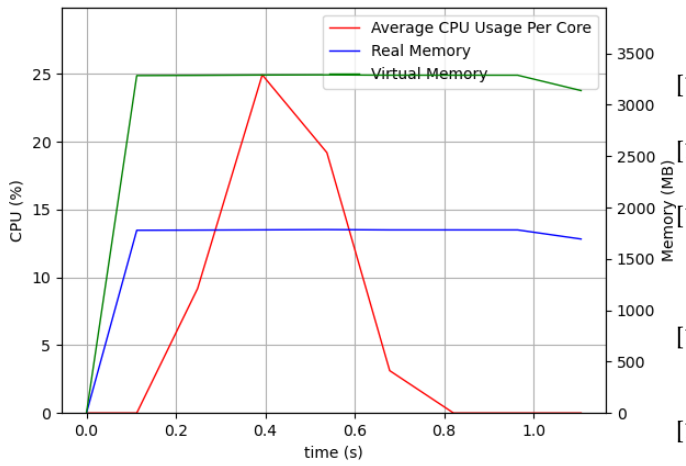

Figure 16: Gradle No Change Building Calculator

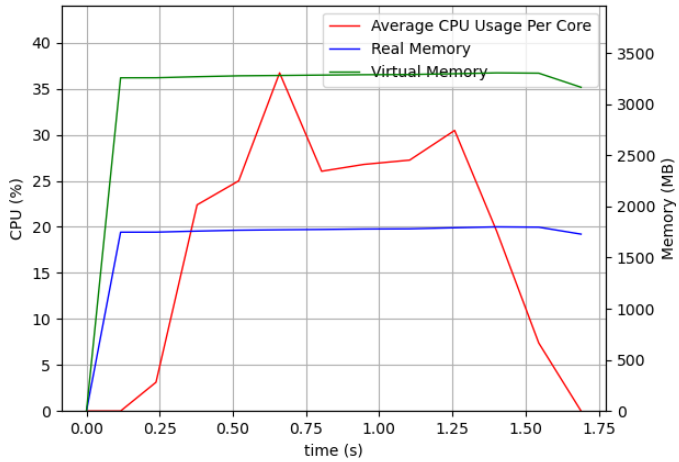

Figure 17: Gradle Change All Modules Building Calculator

[6] (Oct. 2020). "Depgraph-maven-plugin," [Online]. Available: https://github.com/ferstl/depgraph-maven-plugin.

[7] (Aug. 2019). "Gradle-dependency-graph-generator-plugin," [Online]. Available: https://github.com/vanniktech/gradle-dependency-graph-generator-plugin.

[8] (Oct. 2020). "Apache maven," [Online]. Available: https://en.wikipedia.org/wiki/Apache_Maven.

[9] (Oct. 2020). "Takari maven lifecycle," [Online]. Available: https://github.com/takari/takari-lifecycle.

[10] (Oct. 2020). "Incremental maven compilation," [Online]. Available: https://github.com/mariuszs/maven-incremental-compilation.

[11] (Oct. 2020). "Gradle," [Online]. Available: https://en.wikipedia.org/wiki/Gradle.

[12] (Oct. 2020). "Gradle vs maven comparison," [Online]. Available: https://stackoverflow.com/questions/21645071/using-gradle-to-find-dependency-tree.

[13] (Oct. 2020). "Bazel faq," [Online]. Available: https://bazel.build/faq.html.

[14] (Oct. 2020). "Bazel (software)," [Online]. Available: https://en.wikipedia.org/wiki/Bazel_(software).

[15] (Nov. 2020). "Building java jvm projects," [Online]. Available: https://docs.gradle.org/current/userguide/building_java_projects.html#sec:java_source_sets.

[16] (Nov. 2020). "Psrecord," [Online]. Available: https://github.com/astrofrog/psrecord/blob/master/psrecord/main.py.

[17] (Nov. 2020). "Psutil," [Online]. Available: https://github.com/giampaolo/psutil.