Open Source Build System Tools Compared: GNU Make, Scons, Apache Ant

Chris J Arges
University of Texas
Austin,Texas,USA
christopherarges@gmail.com

ABSTRACT
Modern software projects involve many distinct components and dependencies that need to be assembled when building a project. Build system tools help this process by creating a programmable system that can perform sophisticated builds with a minimum of commands. In addition, performance improvements such as only building necessary components and parallelizing builds can be introduced that further improve the utility of a build system. In this paper, three popular open source build systems are compared: GNU Make, Scons, and Apache Ant. By comparing these tools side-by-side developers can better select the appropriate tool for the job.

Categories and Subject Descriptors
D.2 [**Software**]: Software Engineering

General Terms
Design, Reliability, Performance

Keywords
Build Systems,Open Source

1. INTRODUCTION
Building any modern, large software project involves taking a vast amount of components and assembling them into a sum greater than its parts. Although great strides have been made to increase code portability and re-usability; the construction of software still requires sophisticated tools to carefully assemble all the pieces. Build systems need to build reliably, resolve dependencies, build only necessary components, be programmable and adapt to various environments, and be easy to use. [10][9]

Because building software is such a fundamental task, there exist many build system solutions. Depending on language or platform, developers may have had a wide range of experience with different tools. Part of the purpose of this paper is to compare and contrast popular open tools across platforms and languages. By looking at many tools, developers can evaluate the best tools for the job.

Build systems can be integrated into the development environment through command line tools or graphical Integrated Development Environments (IDEs). Many contributors to existing projects may be unaware of this existing substrate on which their code flourishes. However, when a project begins and starts gaining momentum the decision to use a proper build system is critical to the maintainability of the project.

Without a build system, a user would have to understand quite a bit about which dependencies were installed on their system for that particular project. Then they would need to pass appropriate information to the compiler such as the location of these libraries and dependencies, where header files exist, and even which compiler flags are appropriate for this system. Compiling and linking code and object files would need to invoked on many files and directories. Finally, any binaries that were created would need to be copied to the appropriate location. After all this, if no mistakes were made, this process would need to be repeated each time something was changed. In addition, even if a small portion of a project code was changed, the entire project might need to be re-built, not just the few object files that were changed. Unless the person hand-building the project knew all the object files that would be affected by such a change, this is too error prone and complex for most human developers. Thus, motivation for a build system has been sufficiently established.

2. EVALUATION CRITERIA
To compare and contrast the various tools used in this paper, evaluation criteria must be established. First a description of the tool is presented to give context of the tool or collection of tools. Next the widespread adoption of the tool is described with some examples of popular projects that employ this build system. Platforms that are supported are presented, with emphasis on more popular platforms. Language support is considered as well, and to what degree are the details and features of this language handled. Key strengths and weaknesses are presented to highlight advantages of this tool, or perhaps areas where this tool may not be suitable. General performance is then discussed as research data permits. Finally usability is examined and then the various methods a typical developer may use to configure and change build parameters. Using these criteria the tools can be compared and differences examined.

2.1 Overview
For each evaluation an overview of that particular tool is presented. This gives general structure of the tool and how it is used. A simple use case is also presented to illustrate usage. Finally, other relevant aspects of this tool are examined.

2.2 Adoption

Calculating the exact percentage of projects employing this particular build system tool is impossible; thus here examples of popular projects that use this particular build system are discussed. In addition, examples where this build system is part of a standard or accepted practice for a particular set of projects is discussed. How long the tool has been in development and latest release show some indication of how established and popular a tool is. These particular facets of adoption are described here to project insight into whom is using the tool.

2.3 Platforms

This criteria examines the various platforms or operating systems that this particular tool can run on. In addition, any dependencies required or special configurations are discussed.

2.3.1 Languages

The computer languages that this particular build system tool supports are explored. Some languages may have more support than others and this must also be examined.

2.4 Performance

Using available data, the general performance considerations of each tool are compared. This may be difficult if there are substantial differences in the types of projects handled. Thus, here we identify some areas where performance has been optimized and perhaps other areas where there are deficiencies. In addition, identify if this system employs known performance enhancements like compiler caching, parallel builds, or distributed builds. Since in many instances developers only change a subset of the code, describe how this tool handles partial builds and granularity of how these changes are tracked.

2.5 Usability

This particular criteria examines how complex it is for a user to configure build parameters and scripts for a particular project. Describe what is necessary to enable a build system to work across platforms or enabling build variants. Does this tool require numerous esoteric parameters to accomplish a simple configuration? Is the scripting syntax based on a widely used language, or perhaps a standard format is used for configuration files? Again, this is more of a qualitative examination, but at least some degree of usability can be ascertained by the analysis.

3. TOOLS EVALUATION

3.1 GNU Make

3.1.1 Overview

GNU Make is based on the original Bell Labs version of Make, and is widely used in the open source world. It has support for many languages and is stable and mature. It has its own syntax for build configuration files called Makefiles, and these files can be highly verbose and complex depending on the scope of the build. Projects that operate in many different environments needs to be configured using a configure script. To address complexity of different environments projects such as Autoconf, Automake, and libtool have been created to make use of Make and extend its functionality. GNU Make is license under the GNU Public License and is open source. [5]

3.1.2 Adoption

Make has been around since 1979 from the work of Feldman at Bell Labs. [8] Through the years different incarnations of this tool have existed. Currently, GNU Make is the most widely used and popular variant. GNU Make was created around 1988 by Richard Stallman and Roland McGrath as part of the GNU project. It has been in development since then and the latest version was released on July 28th, 2010. [5] Make is used in a very large number of high profile open source projects including: the Linux kernel, gcc, binutils, and most of the tools included in a typical Linux distribution.

3.1.3 Platforms

Most hardware platforms that the GNU Compiler Collection (GCC) support, GNU Make can also be run and and target. This includes many variants of hardware for various Unix platforms, Linux, *BSDs, and embedded platforms. In addition there is support for platforms like OS X (darwin), Windows (via cygwin), and DOS. An extensive list of platforms can be found at `http://gcc.gnu.org/install/specific.html`. [4] Support for platforms like Windows are not as robust as it is not a POSIX-like operating system, thus a layer such as cygwin must be provided to give it the proper dependencies for building.

3.1.4 Languages

GNU Make supports the following languages from GCC: C, C++, Objective-C, Fortran, Java, Ada, and Go. In addition many other types of projects can be built including: TeX, lex, lacc, modula-2, and Pascal.[5]

3.1.5 Performance

GNU Make has automatic dependency checking and automatically determines which files need to be re-compiled. It uses timestamps to check if a file has change. Make can be run in parallel by specifying the '-j' option. This option in turn forks the requested number of processes which then complete the build in parallel as appropriate. The memory footprint of Make is relatively conservative and thus it scales well. [13] [9]

3.1.6 Usability

Simple make files are easy enough to program, but care must be taken to understand the file format and syntax. As programs become more complex, or builds are performed on different platforms or environments additional tools may be required to ease the development of the Makefiles such as Autoconf or Automake. The main part of the Makefile is the rule which specifies a target, any dependencies and what commands to be executed to update a target. Thus syntax can be seen in the trivial example given here: [5]

```
main.o : main.c defs.h
            cc -c main.c
clean :
            rm main.o
```

The project can be built with the following command:

```
make
```

3.2 SCons

3.2.1 Overview

SCons is a newer software construction tool started around 2000 originally as a port of the Cons build system. It is written in Python and in turn uses this scripting language for setting up a project's build environment. It strives to accomplish correctness, performance and convenience. It is licensed under the X/MIT License and is a open source project.

It includes support for many languages and runs on any platform that can run Python. It can build in parallel, cache builds, and only build necessary files to improve performance. It has a concept of a Builder object which can used to support different languages or systems. It also has an Environment object which allows for greater flexibility in storing and extending various variables used in construction and executing tools in a large project build system.

3.2.2 Adoption

Many open source projects have adopted SCons as their build system. Ardour, Blender, Csound to name a few. In addition the closed source game Doom 3 uses SCons as its build system. [7] The last release of SCons was November 25th, 2010.

3.2.3 Platforms

SCons is written in Python, and is it's only dependency. Python can be run on popular platforms such as Linux, *BSD variants, Windows, and Mac OS X. In addition Python has ports for AIX,AROS, AS/400, BeOS, iPod, MorphOS, MS-DOS, OS/2, OS/360, z/OS, Palm OS, PlayStation, Psion, QNX, RISC OS, Series 60, Solaris, VMS, and Windows CE. [6] Therefore, SCons can be used with an extremely wide range of platforms with minimal dependencies.

3.2.4 Languages

SCons has built-in support for a wide array of languages by utilizing a very extensive set of modules. Out of the box, SCons supports the following Builders/Tools: Accumulate, Antlr, Archive, AutoConfig, Cheetah, Combine, Corba, CorbaTao, Csharp, CxxTest, DistTar, DistZip, Doxygen, DumpSODependencies, FltkFluid, GCCXML, Gch, GenericSubst, Ghc, InnoSetup, Lyx, Make, Mex, Nasm, Nsis, Ocaml, Protoc, Pyuic, ReStructuredText, Replacement, Rpc, SubstInFile, Swig, UnTar, Vala, CPython, Cuda, DDK, Docbook, EiffelStudio56, EiffelStudio, Gob2, MicrosoftPlatform, NDDS4, NsisScons, PlatformIndependedInstallation, Qt4, WiX. [7] From this list, C/C++, Python, and Java can be used supporting many popular languages. In addition, any new languages can be supported by amending the Python classes to support any other necessary constructs. Thus, SCons can be used in almost any build system.

3.2.5 Performance

If a subset of the code has changed (which would be the more frequent case), then it would make sense to build less of the project's code in order to improve performance. SCons can handle automatic, implicit, and explicit dependencies in a build. Thus, it can figure out how to build a subset of the project such that the entire project is up to date and correct. Object files that do not need to be changed can be cached and used when necessary. Automatic dependencies include changes in tools, command line options, as well as the contents of the input files. Changes in input files are detected using MD5 signature changes. Implicit changes mean if a piece of code includes a header file and that header file is changed, then both will be rebuilt. Dependencies can also be defined explicitly in a configuration file, as well as telling SCons to ignore dependencies. [12]

In addition, parallel building is also supported by SCons. SCons uses a '-j' option much like Make to specify how many jobs should be started. A thread pool is then created with the requested number of jobs and then used to build the project handling dependencies correctly. [12]

Performance numbers were analyzed in a paper by Ludwig HÃd'hne. [9] His paper compared GNU Make performance with SCons. The paper showed empirical evidence that SCons memory consumption scales linearly with the number of nodes. For example with 500 lines of source code around 20 MB of memory was used, while with 4000 lines of code around 115 MB of memory was used. Another study was done by Eric Melski in 2010 comparing GNU Make with SCons. For a project with 50,000 C files a build took 5 hours, while GNU Make built the project in around a half hour. [13] Thus, performance and memory improvements are needed for SCons.

3.2.6 Usability

SCons is written in Python, and user scripts are also written in Python. This is good because if a user is familiar with Python writing scripts should be easy. For newer users, the syntax and configuration parameters aren't too difficult to learn and can be expressed with a fairly minimal set of lines.

SConstruct is the name of the main script which SCons uses to build. It is very similar to a Makefile in GNU Make. Many different built in commands can be then used to handle typical project setups. For instance a simple SConstuct for a single C language file would be written as follows: [11]

```
Program('hello.c')
```

To invoke a build the following command would be used [11]:

```
scons
```

3.3 Apache Ant

3.3.1 Overview

Apache Ant was started by James Davidson in 2000 in order to provide a better tool for building Java projects. Since Ant is written in Java, it can use classes as tools to build projects, instead of creating new JVMs for each part of the build process. One of its primary features is that all build files are in XML format. The build files can be very verbose

and give the user a great deal of options. In addition, there are many add-ons and related projects for Ant that provide extra features. [14] The last stable release was on December 27th, 2010.[3]

3.3.2 Adoption

Apache Ant is widely used in the Java community. If you use the Eclipse IDE for Java, it has built in support for Ant making it a simple choice for Java developers.

3.3.3 Platforms

Ant can run on many different platforms including: Linux, Solaris, HP-UX, Windows Platforms, OS/2 Warp, Novel Netware 6, OpenVMS, Mac OS X. The most used platforms are Linux, Mac OS X, Windows XP, and Unix. The only dependency required for Ant is a proper Java Development Kit (JDK). [14]

3.3.4 Languages

Java is the primary language that Ant supports. There are extensions available for Ant to build C, C++, and Fortran projects, but a proper toolchain and ant-contrib must also be installed. [1] In addition there are other contributed packages for added support of other languages and frameworks such as Python and .NET. [2]

3.3.5 Performance

Ant doesn't have the same capabilities as GNU Make or SCons in that it cannot automatically parallelize or do dependency checking. It must rely on the user to explicitly show which dependencies exist for a given project, and how those dependencies should be handled. For parallel tasks, these must be explicitly defined in the build XML file. Because of these shortcomings compared to other build systems there is a related Apache project called Ivy (`http://ant.apache.org/ivy`) that handles dependency checking, caching, as well as other performance considerations [3]

3.3.6 Usability

Ant is a very flexible and extensive build system. Many parameters can be specified using its XML format build files. However, one must understand what different options are available to create a well working build file. Ant can be easily extended, and thus there exist many projects extending functionality of this project.

Here is a simple example of a build file from the Apache Ant manual. [3]

```
<project>
    <target name="clean">
        <delete dir="build"/>
    </target>
    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>
    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/HelloWorld.jar"
               basedir="build/classes">
            <manifest>
                <attribute name="Main-Class"
                  value="oata.HelloWorld"/>
            </manifest>
        </jar>
    </target>
    <target name="run">
        <java jar="build/jar/HelloWorld.jar"
                fork="true"/>
    </target>
</project>
```

To compile this project the following commands are executed: [3]

```
ant compile jar run
```

4. CONCLUSION

4.1 Comparison

4.1.1 Adoption

Each of the three tools compared has its own niche. GNU Make is probably the most widely adapted and established tool for C, C++ open source projects. SCons is more obscure, but its real niche is in more python related projects. Apache Ant is a standard build tool for Java projects and has a wide community of support to extend and provide additional features. All tools are being actively developed, although the latest releases are over a half-year old as of this writing. In this category, all three tools tie as they have a niche for which they perform best, despite having some overlap.

4.1.2 Platforms

Platform support is relatively easy for SCons because it is written in Python which has been ported to many different platforms. Ant runs on Java, which also has a wide host of platforms which it can run on. GNU Make supports a wide set of platforms, but has more dependencies. For instance running on Windows requires an extra layer, whereas SCons and Ant can run unmodified. Thus, Ant and SCons have a more portable solution.

4.1.3 Languages

Here GNU Make and SCons support the widest set of languages; however Ant provides the most solid support for Java. Thus Make and SCons have the advantage here.

4.1.4 Performance

Direct comparisons of Ant with other build tools were difficult to find and this could be an area of future work. Fundamentally each have three different models of executing compilation. Make uses a per process model to fork many process to complete the compilation. Scons uses a thread pool to create worker threads that will complete the compilation. Ant uses threads as well, but from within the JVM itsself to complete the build. GNU Make and SCons provide automatic parallization of the build, while Ant requires this to be explicitly done.

SCons has the strongest form of 'correct' building in that it will identify the best subset to compile and thus minimize

the amount of actual compilation that needs to be done. However; this comes at a price as calculating the MD5 sum and tree is somewhat complex. GNU Make will build a decent subset using timestamps, but may occasionally rebuild files that have been updated, even if nothing actually changed in them. This is a good tradeoff as it seems to be overall quicker than building with SCons as shown in the study done by Eric Melske. [13] Ant doesn't have implicit dependency checking, but the user must actually specify every dependency on when a dependency should be rebuilt; however, projects such as Ivy are filling the void and providing more robust tools.

Another important factor is scalability. While the Melske comparison showed that memory usage in large projects increases linearly, Make only requires a modest addition of memory when source files are increased. A similar study of Ant was not found, and this is something that would be nice in future work.

Between Make and SCons, performance and scalability seem overall better with GNU Make. There isn't enough data to compare Ant with these tools; however many of the performance enhancements present in Make and SCons were not available in Ant. It would interesting to see how extensions of Ant improve functionality and performance and compare this with other build system tools.

4.1.5 Usability

Each tool has a different philosophy of how a build configuration file should be set up. GNU Make uses a specialized format defined in the late 70's and continues to use this syntax. Makefiles can be very verbose and long for large projects, but tools like Automake help alleviate some of the complexity. SCons uses a full fledged scripting environment, Python, in addition to specialized functions and classes to provide its configuration. Its files are small and compact, yet powerful. Finally, Ant is greatly concerned about format and uses XML to express its detail oriented configuration files. Many projects have been started around Ant to extend its functionality and give it more powerful commands. Invocation of a build for any of these projects are trivial and of equal complexity.

Overall, SCons provides the most compact and powerful format. In addition SCons makes it somewhat easy to extend functionality. Ant's syntax is very verbose and requires XML-compliant files. This may be difficult for some, yet important for others. GNU Make is somewhat difficult to achieve more complex tasks easily. However since it has been established and mature, many examples of projects exist to aid in developing a custom build system with Make. Qualitatively, SCons provides the most robust yet concise format for configuration.

4.2 Summary

Here is a summary of the comparison. A star indicates that tool is the best in that criteria. A question mark means not enough data was available for evaluation.

Factor	Make	Scons	Ant
Adoption	*	*	*
Platforms		*	*
Languages	*	*	
Performance	*		?
Usability		*	

4.3 Future Work

Comparing performance between the various build systems (especially Ant) using a synthetic data set would be critical to many developers considering a build system for their new project. Saving a developer a minute on a build can be a huge improvement of productivity; thus the value of performance is evident.

4.4 Final Thoughts

I selected this topic because of my interest in learning about a wider set of build tools. In the course of my own work, I primarily use GNU Make; however, I wanted to know what other systems were out there and what they had to offer. Although the tools really have their own strengths and niches, the foremost important thing would be choosing the right tool for a new project. From these results I would choose GNU Make for a C or C++ type project, choose Ant for a Java project, and SCons for any other language. After these factors, Performance and Usability are the most important things when comparing two tools which can be used for a particular project. As a developer whom constantly re-compiles portions of code in a large project, minutes can make a huge difference in productivity. Because of SCons usability, I would choose it for a personal project or where performance wasn't as much of a factor. Overall these tools are still improving, and the projects around Ant and SCons will make these tools worth checking into in the future.

5. REFERENCES

[1] Ant-contrib tasks.
[2] Apache ant - apache ant™ libraries.
[3] Apache ant™ user manual.
[4] Gcc, the gnu compiler collection - gnu project - free software foundation (fsf).
[5] Gnu 'make'.
[6] Python documentation index.
[7] Scons wiki.
[8] S. Feldman. Make—a program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.
[9] L. Hähne. Empirical comparison of SCons and gnu make. 2008.
[10] C. Julien. Architecture and design construction and deployment (class notes).
[11] S. Knight. Scons user guide. *Python Software Foundation*.
[12] S. Knight. Scons design and implementation. In *Tenth International Python Conference*, 2002.
[13] E. Melski. A second look at SCons performance.
[14] J. Tilly and E. Burke. *Ant: the definitive guide*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002.