Großer Beleg

Empirical Comparison of SCons and GNU Make

Ludwig Hähne

June 26, 2008

Technical University Dresden
Department of Computer Science
Institute for System Architecture
Chair for Operating Systems

Professor: Prof. Dr. rer. nat. Hermann Härtig
Tutor: Dipl.-Inf. Norman Feske
Dipl.-Inf. Christian Helmuth

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 26. Juni 2008

Ludwig Hähne

Abstract

Build systems are an integral part of every software developer's tool kit. Next to the well-known Make build system, numerous alternative solutions emerged during the last decade. Even though the new systems introduced superior concepts like content signatures and promise to provide better build accuracy, Make is still the de facto standard.

This paper examines GNU Make and SCons as representatives of two conceptually distinct approaches to conduct software builds. General build-system concepts and their respective realizations are discussed. The performance and scalability are empirically evaluated by confronting the two competitors with comparable real and synthetic build tasks.

Contents

List of Figures

1 Introduction

The choice of the right build system for a certain software project is subject of an emotional debate [McC03]. However, there is little data available to back these arguments. How do these systems actually perform when confronted with the same nontrivial task? A major problem is the migration cost which makes it hard to generate evaluation data for reasonable sized projects. Ideally, one would like to know in advance if it is feasible to employ a certain build system. Furthermore, one may want to move the software project to a different build system to escape the shortcomings in the existing system.

This work tries to identify the benefits and drawbacks of two main players in the field of build systems: GNU Make and SCons. The focus lies upon the core features of a build system. Only occasionally features that mainly serve convenience are mentioned. I hope to shed light on the practicability and performance in real-life scenarios, beyond the scope of trivial programs.

Today's build systems must be up to the task in at least four categories: Convenience, accuracy, performance, and scalability. Of course, convenience is a very subjective category and is assessed differently based on the experiences and expectations of those who have to judge. Hence, this work will focus upon the fields which can be measured. It's important to know the accuracy limits of the employed build system to circumvent resulting problems. Performance and scalability is a correlated domain and also depends on the way the respective tools are used.

The next chapter will give an overview over the state of art and general concepts in the domain of build systems. Afterwards, Make is presented as the leading build system through the last three decades. SCons was chosen as the competitor because its design varies greatly from Make and introduces a lot of new concepts. Furthermore, it was continuously developed over the last eight years and is already used by open source and commercial software vendors to build various medium-sized and even a number of large projects.

An existing GNU Make based build system was ported to SCons with the goal of replicating the existing features. The build performance for various use cases is evaluated and compared. A focus is laid upon parallel build performance. This field is of growing importance as even today's commodity hardware is equipped with multiple processing cores.

Finally, a synthetic project generator was conducted to get an idea about configuration-independent build-system performance. The scalability is evaluated using synthetically generated projects with arbitrary complexity.

2 Background

In this chapter, general build-system concepts will be discussed. The design goals will be examined. Afterwards, the theoretical foundation of a build system is presented followed by a discussion of common build system features. Finally, a brief look is taken at the build systems available at the time of this writing.

2.1 Design Goals

A software build system should have at least four properties. It should be *easy to use*, deliver the results *fast*, make *correct* decisions, and it should be able to cope with increasing demands and growing software complexities.

2.1.1 Convenience

The usefulness of a build system is primarily determined by the degree of helping the developer to achieve his goals. Convenience is a broad field and depends on the amount of supported features, the user interface convenience, the transparency and the extensibility.

2.1.2 Correctness

A vital feature of a build system is that it always makes the right decisions. If it cannot guarantee a correct behavior even in corner cases, the developer will start to mistrust the build system and probably sacrifice performance. A classic case is discussed in [Mil97] where full project rebuilds are sometimes necessary to accommodate the problem of incomplete dependency graphs. However, it might not be possible or feasible to guarantee correct behavior under all conditions. At least, the cases where the correctness cannot be assured should be known and understood by the developer.

2.1.3 Performance

Performance means primarily how fast the build system can perform a requested operation and deliver the result. Moreover, the memory allocated during the build determines the system's performance. Most of the processing power is spent in the tools that are called by the build system. Therefore, the biggest performance gain can be accomplished by minimizing these invocations. Performance is in some ways orthogonal to the correctness requirement, as more elaborate techniques induce a greater processing and memory overhead. However, they might pay off in reducing redundant computations.

2.1.4 Scalability

Software systems are constantly growing in size and complexity. Different platforms need to be supported. Furthermore, many different tools are involved in the build process. A scalable build system can cope with very complex software systems and be adapted to new requirements.

2.2 Software Rebuilding

A build system differs from an imperative program, like a shell script that invokes the compiler, in being goal driven. The user requests a certain target and the build system figures out how to build it.

2.2.1 Dependency analysis

In software build theory, the dependency graph is used to decide whether software modules need to be rebuilt. In theory, a module only needs to be rebuilt when the build command is modified or an imported entity changes in a way so that the generated outcome differs from the one prior to the update. However, that relationship is difficult to model and time-consuming to examine. Therefore, more coarse granularities are chosen in practice. The most widely used operate on the file level. In this case the nodes in the DAG are file names and dependencies can only be modeled between whole files.

2.2.1.1 File signatures

If the dependency model uses file granularity, it must be decided when a file has changed with respect to the dependency graph. Hence, when to rebuild the modules that depend on it. A classic approach used by Make is to take the files' timestamp as a reference. Another way is to compare the files' content, e.g. by checking its hash value. Whereas timestamp-based signatures are simple and more efficient at first sight, they impose a number of problems. First, causal order is deduced from imperfect timestamps based on physical clocks. While this may not be a problem on isolated single-processor machines, where the build process and the files all share the same time source, it puts the correctness of the build into question on distributed and parallel environments. The timestamp approach may also cause unnecessary rebuilds of large parts in deeply nested dependency trees, e.g. if a comment update in a header file triggers the recompilation of an object file producing an identical result. A content-based signature model can stop here while timestamp-based systems need to rebuild all subsequent files which depend on the object file. On the other hand, file content comparison or hashing are more computational challenging than timestamp comparisons, therefore hybrid methods also have been developed, which require a minimum time lag, chosen to bypass real-world clock offsets, or otherwise resort to file content comparison.

Simple timestamp comparison can also lead to wrong results. If the source file is restored from a backup archive, the timestamp may change to an older date. A stateless

build system would not rebuild in that case. This problem can be solved by tracking the timestamp of the source file which is used to build a target and trigger a rebuild if it changes.

Various improvements and variants were discussed, for example to eliminate source code comments before performing the content comparison [Bor89, ATW94].

2.2.1.2 Fine grained dependencies

The drawback of a model that regards files as opaque entities is the potentially huge amount of redundant recompilations. For instance, if a rarely used function of a core library is altered, all targets who link to the library need to be rebuilt, even though few of them ever call the altered function.

Smart recompilation [Tic86] attempts to minimize recompilations by refining the dependency relationship. Each dependency context is extended to include individual items (i.e. symbols). Before a node is rebuild, the *change set* is matched with the *reference set*. Only if the intersection is not empty, the recompilation is triggered. This idea was later refined to further reduce recompilation efforts [SK88, Tic88].

The downside is the domain-specific nature of these approaches, which make them difficult to implement in general purpose build systems.

2.2.1.3 Dependency declaration

The dependency relationships which make up the DAG need to be described. One possibility is to require the manual declaration of all nodes a specific node depends on. On the contrary, the build system could extract all dependencies.

The advantage of the first approach is that the build system does not need to incorporate domain-specific knowledge. On the other hand, explicit dependency declaration is inconvenient and error-prone. Implicit dependencies are already described inside the source files. The canonical example are `include` statements in C sources.

Most modern build systems support automatic extraction of implicit dependencies for a variety of languages. Either the compiler is used directly or language-specific tools are incorporated into the build system.

2.2.1.4 Dependency types

Real Dependency Describes a precedence relationship. Before the node can be built, all nodes it depends on need to be generated. The node needs to be rebuilt if any of its dependencies change.

Order-only dependency Describes a precedence relationship without the need to rebuild after a change.

Intermediate nodes A temporary node which is created implicitly by a chain of build commands.

Resource constraints To prevent parallel execution of build commands, shared resources can be modeled as common nodes.

2.2.2 Build infrastructure

An often ignored topic is that changes in the build infrastructure might entail changes in derived files. It is easy to understand that different build flags will probably produce different target files, for example if debug symbols are included or omitted. However, the compiler itself plays also an important role. If a different compiler is used in a subsequent build, a derived file might need to be regenerated even neither the source nor the build command changed. Therefore, the build tools should be modeled as an implicit dependency.

Nevertheless, few build systems include the build infrastructure in the dependency analysis. Many problems arise when trying to comprehend which tools may have an impact on the outcome of a build. For example, the GNU compiler collection comprises many tools which are involved in a build process. However, few of those may cause a derived file to change due to modification of its internals. Most build systems assume the build infrastructure to be fixed across subsequent builds and rely on the developer to force a clean rebuilt when a vital part of the build infrastructure has changed.

2.2.3 Command scheduling

Once the DAG is set up, the build commands have to be executed. The DAG describes a precedence relationship between transformation commands. The build system has to make sure that commands are executed in the proper order. To generate a total order compatible with the partial order relation defined by the DAG, *topological sort* can be used [CLRS01].

Most build systems provide a facility to execute commands in parallel. However, some compilers interfere with each other or do not allow to execute in parallel. This can be modeled with *resource constraints*. Developers need a way to tell the build system about such mutual exclusion cases.

The goal is to minimize the *makespan*. This is the problem of scheduling task graphs to a set of available processors. Although this is mainly of interest on multiprocessing platforms, uniprocessor builds might also benefit by executing different tasks when stalling on I/O operations. Baalbergen described *virtual processors* which process jobs non-preemptively. Computing an optimal schedule with regards to a minimal *makespan* is a NP-complete problem [Baa88]. A number of different heuristics were proposed to solve the multiprocessor-scheduling problem [GGJ78]. However, these algorithms make a number of assumptions which have to be relaxed in the context of build-system scheduling. In build systems, (1) jobs are not independent (precedence constraints); (2) task execution times are not known in advance; (3) the DAG might be refined when executing commands, specifically when implicit dependencies where extracted from generated source files; and (4) mutual exclusion constraints may prohibit running certain commands in parallel (resource constraints). Therefore, a single scheduling solution for the whole build is not applicable in general.

The most common scheduling algorithm incorporated into build systems is a variant of FCFS (First Come First Served). The scheduler places all ready jobs into a queue

which is processed by a set of processors. Some build systems also allow to define upper load limits.

2.3 Build System Features

2.3.1 Partial compilation

Large software projects contain a big number of modules, programs, mandatory and optional parts. The user shall be able to specify the parts of the project that must be built. These can be directories, object files, programs, individual or sets of modules. Ideally each node in the DAG can be built by passing its label to the build system.

2.3.2 Variant builds

Building variants is the ability to produce different targets from the same sources. The canonical example are debug and release variants where the first contains debug information and the latter is optimized. Another common case is building for different target architectures. Different variants can either be build on demand or be generated side-by-side. The latter requires multiple destinations for the products of the same sources.

2.3.3 Repositories

The repository concept was introduced to allow developers implement new features without touching the original source tree. A repository is a replication of the source directory structure. Multiple repositories build a single logical tree where an additional repository represents an overlay to its predecessors. A repository might supplement new files or cover existing files.

There are a number of conceptual problems with repositories. Build rules are usually part of the directory tree and are also affected by repository overlays. Therefore, the build tool must be aware of the repositories to be used before it can select the appropriate build rule definition. This can be done by command line parameters or external configuration files which is rather inconvenient. Another problem is that the overlaid files totally cover existing files with the same name in other repositories.

2.3.4 Configuration

Configuring the software system comprises two main features. One gives the user the possibility to configure the software system based on his preferences. The other is to (automatically) adapt the software system to the target platform. The build system can support this by passing configured flags to the compiler or generate code based on the configuration.

2.3.5 Compiler caching

A build is comprised of a large number of intermediate files being generated. Much time can be saved when these files are cached and reused in later or other builds. For

example, an object file might be linked to multiple targets. In case this object file is cached while building one target it does not have to be regenerated when trying to build the other target later.

Caching can also be employed in multi-developer build environments where a central build cache is used to store shared derived files. Care must be taken to assure that derived files have been generated with the same commands. Accordingly, the build cache should also contain all necessary build information. A compiler cache can also help to store multiple versions of the same file. For example, an object file is stored with and without debug symbols.

The compiler cache can be integrated or transparent to the build system. In the latter case, the compiler is wrapped by the cache manager. *compilercache* [Thi03] and *ccache* [Tri04] act as preprocessors for C and C++ compilers. The preprocessed source file is hashed and matched with the cached variants of the file. If the file is already in the cache it is returned, otherwise it is compiled and added to the cache.

2.3.6 Parallel builds

Multicore and Multiprocessor architectures are widespread and should be exploited to speed up the development process. The build system must be able to schedule individual build tasks to run concurrently while not violating dependencies. A number of issues was already discussed in 2.2.3. Amdahl's law states that the maximum speedup is determined by the serially executed fraction [Amd67]. If P is the fraction of the total runtime which can be executed in parallel, and N is the number of processors, the maximum achievable speedup is limited by the non-parallel executed fraction:

$$\lim_{N \to \infty} \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{1-P}$$

This is worth consideration because most build systems' operations are serially executed while the command execution is parallelized. Therefore, the build system performance imposes a limit on the maximum performance achievable by parallel execution.

2.3.7 Distributed builds

The idea behind distributed compilation is simple. In a network of hosts which are dedicated to build a large number of source files, one host assigns build jobs to other hosts configured as build daemons. For distributed builds to be beneficial in terms of overall performance, the assignment of build jobs, transmission of the needed data, and retrieving of the results must be faster than carrying out the build job locally. Therefore the infrastructure performance parameters play a key role in deciding whether a distributed build architecture is feasible.

Distcc [Poo03] is an approach that fits very well into existing build systems and can indeed be used with both Make and SCons. The preprocessed sources are transferred to build daemons for compilation. One problem are the potentially different build environments on the build daemons. As the build system does not know about the remote

(and unknown) build environments involved it cannot solve the problem discussed in 2.2.2.

2.4 Build Systems

In this section, we will take a brief look at a number of existing build systems. Make and SCons will be examined in later chapters.

2.4.1 Jam

Christopher Seiwald designed Jam in 1993 to replace Make as the commonly used build system [Sei94]. Jam has its own syntax that supports control-flow statements among other features and which simplifies the description of software builds. Implicit dependencies are extracted using regular expressions. The built-in rules primarily support building C and C++ software.

Several different variants of Jam were developed, most notably *Boost Jam*, *FT Jam*, *KJam* and *Autojam*.

2.4.2 Apache Ant and Maven

Apache Ant [Ant06] is a Java-based build system that uses Java classes instead of shell commands. The build description files use a XML-based syntax. Ant is the preferred tool to conduct Java builds. It contains a lot of domain knowledge and built-in tools which are useful to build Java projects.

The verbosity and complexity of the Ant XML build definitions led to the development of other build and software project management tools like Maven [Mav08]. Maven adheres to the concept of *convention over configuration*. This simplifies the user supplied files but restricts the software structure to its project model.

2.4.3 Rake and Rant

Rake [Wei04] was designed to be a Ruby based alternative to Make. The build description files are implemented as Ruby scripts. It supports pattern rules and flexible file lists among other things, but depends on manual dependency declaration and timestamp signatures. Rant [Lan05] is another Ruby based build system that also supports content signatures and implicit dependency extraction for C/C++ sources. Rant also supports packaging and testing for Ruby software projects.

2.4.4 Overview

Build System	Since	Language	Signature	Implicit dep.
Make	1979	own	Timestamp	no
Jam	1993	own	Timestamp	yes
Bras	1996	Tcl	Content	yes
Cons	1997	Perl	Content	yes
Apache Ant	1999	XML, Java	N/A	N/A
SCons	2001	Python	Content	yes
Makepp	2002	Perl, Make	Content	yes
Rake	2003	Ruby	Timestamp	no
Rant	2005	Ruby	Content	yes
Waf	2006	Python	Content	yes
GNU Autotools	1994	M4, Make, Shell	Timestamp	yes
CMake	2000	own	N/A	yes

As can be seen, most of the more recently developed build systems use scripting languages, instate content-based signatures and provide facilities to extract implicit dependencies.

2.5 Summary

General build system concepts have been presented. Convenience, correctness, performance and scalability were identified as the main design goals. An overview was given about the available build systems and common features.

3 Make

In this chapter, the Make build system will be examined. After a short excursion to the development history, GNU Make is studied as the de facto standard Make variant in use today. With Autotools and CMake, two important Makefile generators are discussed.

3.1 History

Make has undergone 30 years of software development. Though it has been optimized and extended, its fundamental mechanisms still follow Stuart Feldman's original proposition [Fel79]. During the years, a number of different Make variants were developed. Grosskurth [Gro07] gives a concise overview of the most interesting branches and developments.

With the advent of upcoming multi-processor systems, the possibilities of incorporating parallelism in Make was investigated by many researchers. Baalbergen [Baa88] examines scheduling algorithms and proposed a system which employs parallelism transparent to the user.

Miller [Mil97] demonstrates the problems of recursive usage of Make. It discusses that a whole project Makefile approach eliminates the problems of incomplete dependency graphs and is still effective enough to be used in large projects.

3.2 GNU Make

GNU Make is probably the most popular Make variant in use today. The idea of portability has shifted from writing Makefiles understood by different Make variants, towards a portable Make build system. This allows to exploit the more advanced features of the build system, and not be limited to the least common denominator of partly incompatible variants.

3.2.1 Build Definition Syntax

Make uses a simple syntax to specify targets, dependencies (called prerequisites) and transformation commands. Build definitions are expressed in a domain-specific language.

```
target : prerequisites
        command
```

The user requests the target to be built by invoking Make with its name. Make checks whether the target's timestamp is newer than all of the prerequisites' timestamps. If it isn't, the command is run to build the target from the source files.

Targets can be annotated with additional information by listing them as prerequisites of special targets. .PHONY targets build unconditionally, .INTERMEDIATE files are cleaned after the build, .LOWRESOLUTIONTIME deal with inaccuracies of timestamps. There are more described in [SMS06].

There is a syntax to specify Make rules and a different syntax to specify shell commands. The Make syntax provides variables, text transformation support, conditional parts and pattern rules.

3.2.2 Repositories with vpath

To support the repository concept, GNU Make has the VPATH facility. It comes in two flavors. A VPATH variable lists directories which are searched when looking for prerequisites and targets. Another more elaborate method assigns filename patterns with certain directories. The latter can be chained to look for certain files in a determined order.

When targets are out-of-date, they are rebuilt locally even if the prerequisites are located in a remote repository referenced in the VPATH variable.

3.2.3 Implicit rules

Make rule syntax provides implicit rules for built-in transformations. Implicit rules are applied whenever a rule without command line is specified or a file is referenced for which no rule exists. The rule is selected from a catalogue based on the target file pattern. From this pattern, the prerequisite file is derived. Implicit rules can be *chained* to build a file in multiple steps.

The commands invoked by implicit commands depend on a fixed set of variables. These can be altered in advance to select different compilers or change the passed flags.

3.2.4 Automatic dependency extraction

To extract implicit dependencies out of source files, third party tools are needed. The GNU Make manual recommends to emit dependency files containing Make rules for each target file. These are included into the Makefiles.

3.2.5 Command scheduling

If no parallelization parameters are given, Make executes the commands one after another in the order determined by the precedence relationship. The `-j`*x* parameters controls the number of *job slots* used to execute commands. Parallelization can be prohibited for the whole Make instance with the .NOTPARALLEL target. There is no facility for mutual exclusion of commands.

In recursive configurations, Make enforces this limit even across multiple Make instances running in parallel. The Make instances itself do not allocate a slot, so that there are slots left to execute the actual commands.

Furthermore, Make supports load limiting. If the system is above the given load threshold, no additional job is executed.

3.3 Makefile Generators

Writing Makefiles involves some recurring tasks like platform configuration and implicit dependency extraction. Attempts were made to automate the process. Most notably, the GNU Autotools and later CMake became commonly used on top of GNU Make.

3.3.1 GNU Autotools

In the 90's, a number of inconveniences of the Make program were addressed by an upcoming toolchain, which was later referred to as *Autotools*. The first among those tools was Autoconf with the goal to reduce the redundant platform configuration issues, which many open-source projects of that time had to face.

Autoconf [MED02] lets the user conduct a number of tests expanded to a shell script from M4 macros. Based on the result of these tests, user configurable actions are performed. This allows to adjust the build environment to different platforms, ensure that the requirements are met, the proper libraries are installed and configure software components based on user input. Zadok quantifies the benefits of using Autoconf and arguments that it helps to improve portability and to reduce the size of the code base [Zad02].

The advantage of this approach is the minimal requirements imposed on the target platform. As the configure (shell) script is generated on the developers machine, the platform where the software is to be deployed only has to offer a compatible shell, which excludes Windows for that matter.

Automake [MTDL08] minimizes the size of the control file the developer has to specify. Build needs are specified in `Makefile.am` file from which a full fledged Makefile is deduced with a load of canned build rules for installation, testing, and deployment. Automake also transparently extracts implicit dependencies.

The downside of Automake is that it forces projects to apply the GNU mindset to organize software builds. Another hitch is that Automake does not provide the full flexibility of GNU Make.

Libtool [MOTV08] helps to build and deploy shared libraries in a portable way. One problem of Make is that it has problems to deal with variable suffixes of target files. Libtool also aids library versioning and linking to shared libraries.

Over the years, a lot of platform knowledge has been assembled in the M4 macro database. This is one of the main reasons, the Autotools are still in widespread use today, despite the reliance of otherwise replaced languages like M4. Autoconf is, next to Make, the mandatory and probably the most useful part of the toolchain. At least this is indicated by projects like Apache, Python and others who replaced Automake control files by direct Makefiles configured through Autoconf.

Although portability is the main design goal of the GNU Autotools, its support for common platforms like Windows is amiss.

3.3.2 CMake

CMake is a rather new development that has received significant attention lately when it superseded the Autotools based build system of KDE [Neu06]. CMake differs from Autotools in that it isn't bound to GNU Make as the build system back end. For example, on Windows platforms Visual Studio can be used to perform the actual build.

CMake describes itself as a "cross-platform build system generator" [Kit08]. Software builds are specified in the CMake syntax. These specifications are transferred to control files for one of the supported build systems on the target platform. CMake provides features comparable to Autoconf for configuration.

3.4 Summary

GNU Make's features have been presented. Make provides a simple syntax to describe build dependencies between files. A number of tools evolved to simplify Make's usage. The GNU Autotools and CMake were introduced as examples of Makefile generators.

Makefile generators and wrappers still inherit all the problems and short comings of the underlying build system. Furthermore, the loose coupling makes it more difficult to profile and optimize the build system's behaviour.

4 SCons

In this chapter, the SCons next generation build system will be presented. The design goals and architecture are discussed, concluding with a brief look at optimization techniques and some interesting implementation aspects.

In contrast to the GNU build toolchain, SCons is a totally self-contained tool that does not depend on existing platform utilities. The goal is to provide the user the full power of a modern scripting language, Python in the case of SCons.

SCons has been in development for eight years now. It started as a port of the Cons [Sid98] build system which chose Perl as the scripting language. SCons favors build correctness over build performance.

4.1 Architecture

The key to exploit all the features of the SCons build system is to understand the architecture and the interaction of the different components.

The core functionality of SCons is to define a dependency graph using the SCons API. Actions are specified to transform source nodes into target nodes. Once the DAG is set up, SCons walks the tree, checks whether the nodes are up-to-date by computing the MD5 hash of the file contents. If a node needs to be rebuilt, a job is added to the work queue, which is processed by worker threads, allowing parallel builds.

Implicit dependencies are extracted by scanners which are available for a number of languages. Another important feature is the encapsulation of the environment state in a Python object. This allows to modify, copy, inherit environment variables and isolate different build environments from each other.

What makes SCons stand out in comparison to other build tools, is the object oriented approach to manage software builds. Isolation can be achieved by instantiation. Components can be extended via sub-classing. On the other hand, the API makes build control files more verbose and complex, compared to other build systems.

4.1.1 Node subsystem

The Nodes make up the dependency graph and map the graph nodes to external representatives. In most of the cases these are files, but those also can be directories or Python values.

The idea behind the Node class hierarchy is to separate dependency management from the actual Node representation (i.e. files, directories, values). The dependency management is implemented in the Node base class while the Node representations inherit from this class.

4.1.2 Builder subsystem

Builders are the glue logic connecting Nodes with each other. Actions are associated with extensions. Scanners are associated with source and target nodes. Emitters manipulate the source and target lists. Generators can generate actions on-the-fly. Builders can be chained together which allows multi-stage building. Builders translate its arguments into Nodes and set up Executor objects to perform the execution of Actions.

Actions can execute a command (`CommandAction`), a Python function (`FunctionAction`), or encapsulate a list of other Actions (`ListAction`). The Executor computes the signature of the Action and applies the up-to-date status of the targets being built.

4.1.3 Environments

Construction Environments encapsulate the environment state (i.e. environment variables) and get Builders assigned to it. Environments are used to generate different build variants side by side and assign different build parameters to parts of the system. Furthermore, Environments are the facility of choice to communicate variables to subordinate SConscript files.

Environments can be cloned to create slightly adjusted variants of an existing environment.

4.1.4 Signatures

SCons signatures are content based. The Nodes get a MD5 content signature which is saved to a database along with the modification timestamp and the size of the file. Furthermore, the action objects (or transformation commands) used to build derived files are also hashed. Figure 4.1 on the facing page depicts the signature database of a simple hello world program. Therefore, whenever either the file is modified, the commands or flags change, a different compiler is used, the dependent Nodes will be rebuild.

A Decider function is queried to inspect if a target must be rebuilt. The Decider is called for every dependency of a Node along with its signature from the last build. SCons provides content-based and timestamp-based default Deciders. Different Deciders can be attached to different environments.

4.1.5 Scanner

Scanner objects are responsible for extracting implicit dependencies out of source files. For each source file, an associated scanner is invoked which parses the file and returns a list of files it depends on. A scanner may also process a file and its dependencies recursively.

A C scanner might be implemented by searching for `#include` statements. This is not really accurate, as the scanner would need to implement the full C preprocessor. If the accuracy of the supplied scanner is insufficient, compiler generated dependencies can also be used instead.

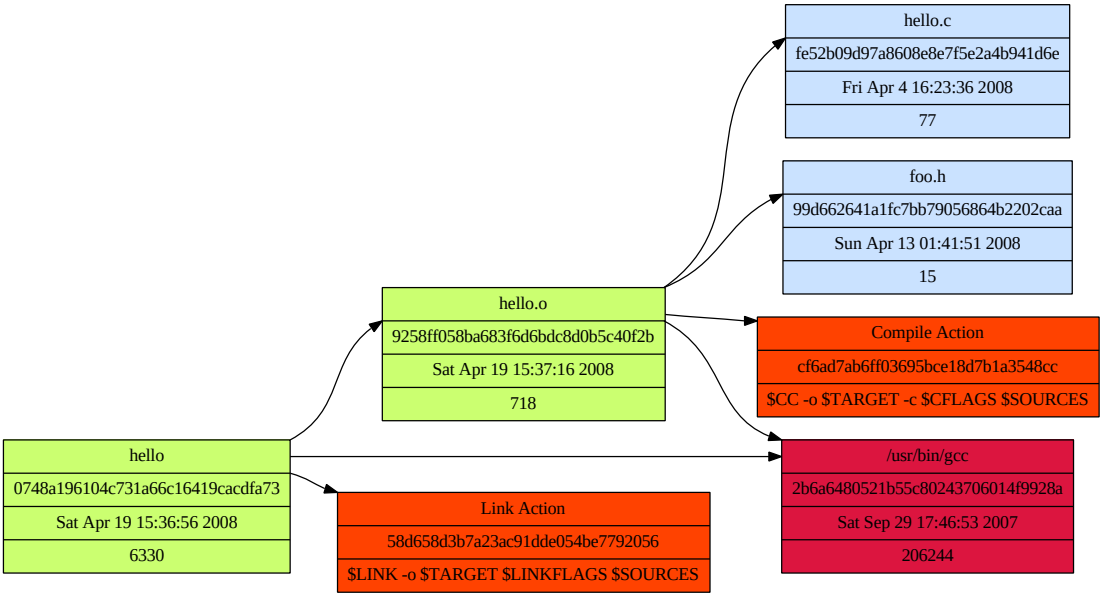

Figure 4.1: SCons signatures example

4.1.6 Tools

Domain specific knowledge is encapsulated in Tool objects. Tools are responsible to detect utilities and prepare the Environment to make use of them. This involves setting up Builders and instating necessary variables.

SCons ships with a number of Tools for different platforms, languages and compilers. The built-in tools are slightly more powerful then the GNU Make implicit rules.

4.1.7 Scheduler

The subsystem that schedules Jobs for execution is called Taskmaster in SCons. The Taskmaster ensures Nodes are evaluated in the proper order and interfering builds are not executed in parallel. In general, the DAG can be converted to a linear list using *topological sort*. However, specific builds may only need to deal with small subsets of the whole DAG. Therefore, the DAG is walked starting from the specified top-level Nodes down to the leaf Nodes of the graph partition.

The Taskmaster runs in a single thread. However, the individual Tasks evaluating the signatures and executing the build commands are run in separate threads.

The Taskmaster which has a runtime complexity of $O(n)$ is invoked for each processed Node. Thus, the overall complexity is $O(n^2)$.

Preventing parallel execution of certain build commands can be achieved by attaching a common Node as a side effect. The canonical use case is the fixed temporary file used by Yacc builders [Baa88].

4.1.8 Configuration subsystem

SCons provides Configure Contexts which should help configuring the build system comparable to GNU Autoconf. It provides some basic checks which adapt a specific Environment to the build platform. The availability of functions, libraries, header files and type definition can be checked. Furthermore, custom checks can be executed. Typically a compiler is invoked on a synthesized source to check for certain features needed by the software system.

4.2 Optimization

The improvements of the build accuracy and the more elaborate design have a significant impact on build performance. Therefore, a number of different techniques were incorporated into SCons to tackle performance aspects from different directions.

4.2.1 Implicit cache

In analogy to the generated dependency files described in 3.2.4, SCons provides a mechanism to reuse already extracted dependency information. Use of the implicit cache trades in correctness for runtime performance. The problem with implicit caches relating to the location of implicit dependencies is discussed in 6.2.2. An approach to overcome this limitation is to also cache all locations where implicit dependencies are not found. Subsequent builds check these locations in the proper order to detect new files which shadow cached dependencies. According to one of the SCons developers, the described technique is going to be integrated in future SCons versions.

4.2.2 Derived file cache

When multiple developers work on a single project, a lot of compilations are redundant across different workstations. The CacheDir mechanism makes it possible to share derived files located on a public accessible file system. For example, if CacheDir references a NFS-mounted file system the built files are stored in this directory. Subsequent builds reuse the cached files. The build signatures serve as the identification.

4.2.3 Interactive mode

SCons has an interesting feature added in version 0.98. The goal is to reduce the parsing time in typical edit, build, test work cycles. Instead of reevaluating the build description files on every invocation, SCons starts a console after parsing those. The console allows to build and clean targets. Interactive mode eliminates the parsing overhead and reduces the startup and response times.

4.3 Implementation Aspects

Python as the build description language offers a lot of possibilities to the build system developer. As SCons itself is also implemented in Python, practically all aspects of

the build system can be adapted to the specific requirements of the target system. Furthermore, Python provides a collection of utilities to tackle performance problems. These come in handy to optimize the build system for large software projects. Most notably, SCons can be run through the Python profiler which reveals how much time is spent in each user and SCons library method alike.

SCons uses metaclasses to implement and profile caches for internal objects. Instead of computing certain objects over and over again, these are stored in class-private caches. Hit and miss statistics provide insight how useful it is to cache the object.

The script user interface is decoupled from the SCons execution engine. Therefore, it is possible to replace the interface with a different front-end which might be more suited for the task at hand.

4.4 Summary

SCons' design goals and architecture have been examined. In comparison to Make, SCons has a more elaborate stateful signature subsystem. It attempts to incorporate all aspects of modern build systems in a single tool, including configuration and build caches. The object oriented approach makes it possible to adapt SCons to different use cases and offers great flexibility to the build system developer.

5 Case Study

In this chapter, the GNU Make based build system of Bastei [FH06] is evaluated and the port to SCons is described. The effort, benefits and shortcomings will be examined. A quantitative comparison follows in the subsequent chapter.

5.1 Build System Design Aspects

The Bastei build system is heavily oriented towards user convenience. Additional features can be added without touching existing sources or build files. This is achieved by exploiting the repository concept and the declarative nature of its build specification.

The build system uses a recursive Make model. To avoid problems with incomplete dependency graphs, the build system splits the project in two logical parts. A part listing all the source files unique to the target built, and another part comprising all shared sources. The latter are called libraries in this context. Whenever a target is built, the build system walks down the dependency chain and rebuilds the needed libraries. To improve performance, the derived library files are cached.

Listing 5.1 shows a `target.mk` build description file which controls the build of a binary with the name *nitpicker*. There is not a single Make rule. The Bastei build system derives the Make rules based on special variables (e.g. `SRC_CC`). The VPATH feature is used to search for files in different locations.

Listing 5.1: Bastei target description example with Make

```
TARGET   = nitpicker
REQUIRES = bastei
LIBS     = cxx env server blit
SRC_CC   = main.cc \
           view_stack.cc \
           view.cc \
           user_state.cc
SRC_BIN  = default.tff

INC_DIR  = $(PRG_DIR)/../include \
           $(PRG_DIR)/../data

vpath %.cc $(PRG_DIR)/../common
vpath %    $(PRG_DIR)/../data
```

Configurability was another design goal. The configuration subsystem is an integral part of the Bastei build system. Platform specific components reside in separate files.

Porting Bastei to another architecture does not require any changes of existing files. Instead, porting is done by *supplementing* new files.

The specialization mechanism allows to add a Makefile that covers the new architecture. When the user specifies this new "specname" in the specialization configuration file, all generic software components will be built for the new platform without any changes of the existing build descriptions.

5.2 Migration to SCons

The intention was to replicate the convenience and style of the build specifications.

SCons repositories were employed to mimic the Bastei repository concept. Those fitted well the requirements and were even less verbose at the cost of some flexibility. The VariantDir mechanism separates the source directories from the target directory. Due to the way one has to specify source files it is rather non-intuitive, but it serves its purpose.

To invoke the build specifications the SCons Glob facility was used, which integrates nicely with repositories.

Finding a way to incorporate the declarative style of the build files and hide the SCons business logic was more challenging. I have chosen a model where these specifications are executed as Python scripts and later parsed to derive SCons rules from them. One problem encountered with this non-standard usage of SCons is that the node creation is deferred. With the original Make build files one can add custom Make rules to the declarative statements for example to execute tests with the compiled target. To overcome this limitation, post-process functions were added which can directly utilize the SCons API. The resulting descriptions alienate from the desired Make look & feel and are perceived as being awkward.

Listing 5.2: Bastei target description example with SCons

```
TARGET   = 'nitpicker'
REQUIRES = 'bastei'
LIBS     = 'cxx env server blit'
SOURCES  = '''main.cc
             view_stack.cc
             view.cc
             user_state.cc'''
BINARIES = 'default.tff'
DIR      = 'src/server/nitpicker'
PATHS    = 'common bastei data'
INCLUDES = DIR+'/include '+DIR+'/data'
```

Listing 5.2 shows the SCons build description file corresponding to the `target.mk` shown in listing 5.1. This is a Python script which is executed in an isolated namespace. Therefore, most variables are represented as Python strings. As the repository-aware Glob facility is not able to search recursively for build description files at the time of this writing, the build descriptions are placed in a central location of the repository. The directory to the source folder must be specified as it cannot be derived from the

description file anymore. The VPATH facility is emulated by a less powerful manual search for source files. However, source types can be derived from the extension and need not be discriminated via an explicit variable.

What actually made some rules easier compared to Make is the possibility to specify Builders in Python syntax. The specific requirements of linking binary files to targets was more straightforward to implement as a Python function and more efficient.

5.3 Comparison

The SCons port has been to designed to resemble the convenience and description style of the GNU Make based system. The user-visible features are very similar in both build systems. However, it was tried to reuse existing SCons concepts like `VariantDir` and `SourceDir` to achieve the desired behavior. The target and library description style covers a lot of build system complexity and business-logic but also sacrifices flexibility in the case of SCons.

The core of the GNU Make based Bastei build system is implemented in 5 Makefiles which sum up to 615 lines of code. The target, library, and specialization build descriptions are distributed over 82 files and contain 800 lines in total.

The core of the SCons-based Bastei build system contains 678 lines of code in 4 files. The build descriptions are scattered over 83 files and contain 677 lines of code in total.

For what it is worth, there is no significant difference in the quantitative LOC measures. Due to the specific requirements of the Bastei build system, the core of the respective build systems need to implement the logic for the library and target concept. Furthermore, the minimal declarative build description requires some wrapping overhead, especially in the case of SCons. This is the main reason for the rather complex core description files.

5.4 Summary

It was possible to migrate most features from the Make based build system to a SCons based variant. The behavior is not totally equivalent and the build description files are not compatible, although similarities exist. It must be noted that it was not possible to perform the migration step by step. A substantial amount of work was needed to recreate comparable behavior and meet the original design goals. The resulting SCons build system meets these criteria.

6 Evaluation

In the previous chapters, GNU Make and SCons were presented individually. The Case Study explored the *convenience* facets and the migration cost to move from Make to SCons. In this chapter, both build systems' supported features will be compared. It is evaluated how the build systems comply to the design goals of *correctness*, *performance*, and *scalability*.

6.1 Feature comparison

GNU Make and SCons have different strengths and weaknesses. Table 6.1 shows what features are supported by the two build systems and how they are instated.

Table 6.1: Feature comparison

Feature	GNU Make	SCons
Partial compilation	Target names	Filenames, Alias
Variant builds	-	VariantDir
Repositories	vpath	SourceDir
Configuration	-[1]	SConf
Implicit dependencies	-	Scanners
Parallel builds	yes	yes
Load limitation	yes	-
Mutual exclusion	-	SideEffect
Compiler caching	-[2]	CacheDir
Distributed builds[3]	-	-

[1] Supported when used with Autoconf or CMake
[2] Supported when used with CCache
[3] Supported when used with Distcc

6.2 Limitations of Build Correctness

One has to differentiate two types of build system decision errors. On the one hand, a build system might build too much, sacrificing performance. On the other hand, a change might go unnoticed and no rebuild is issued. The latter case might be troublesome. A developer might spend precious time, hunting a bug that's already fixed, if it had only been built correctly.

A number of factors are discussed leading to wrong build results. The reasons are identified for common cases of wrong decisions and how problems can avoided.

6.2.1 Signatures

The fundamental challenge of a build system is the decision whether a file needs to be rebuilt. GNU Make's stateless timestamp approach does not rebuild in the following cases, even it deems necessary:

- A source file is restored from a backup archive and receives an older timestamp.
- Environment variables change, e.g. the compiler flags.
- Different transformation tools are used, e.g. a different compiler.

Conversely, build systems frequently build too much and sacrifice performance, for example when changing comments in source files. The timestamp-based signature system will rebuild all downstream target files which depend on the generated object file.

SCons signature subsystem keeps a lot more information than what Make can extract out of file timestamps.

6.2.2 Transient Headers

An inherent challenge in C/C++ software projects is that of including the right header files. There is a list of locations where a header file may be located. The build system needs to track the proper header files to decide whether a header change triggers a recompilation.

The GNU Make manual recommends a dependency cache to store implicit dependencies (see 3.2.4). There are a number of problems with keeping the cache up-to-date. If a once included header file is removed (and therefore no longer part of the DAG), it is still stored in the cache. GNU Make introduced the no-prerequisite targets to overcome this issue.

Listing 6.1: No Prerequisite Targets

```
foo.o: foo.c foo.h
	$(CC) $(CFLAGS) -c $< -o $@
foo.h:
```

In Listing 6.1, `foo.o` is rebuilt when `foo.h` is deleted and no longer a dependency.

Even more challenging is shadowing of header files. A modified header file is placed in a location which takes precedence over the location where the cached file is stored. Make does not trigger a rebuild because it still considers the dependent targets up-to-date. Only a manual rebuild of the implicit cache solves this problem.

6.3 Performance Comparison

6.3.1 Use Cases

To provide meaningful data, real-life use cases are used to illustrate the build system's performance on every day's build tasks. The most fundamental task is probably a *full project build*. This is not only interesting for developers but also for users who might install a package from source code. It is expected that the different build systems have comparable performance as this should be determined by the commands executed (compiler, linker, etc.). Beside the total build time, the number of processes spawned is measured and the time spent parsing build rules.

Another common task that should give an indication of the raw parsing and dependency analysis performance, is an *up-to-date check* when no commands need to be executed. That means after a full project build, another full build is issued which should just report that the derived files are up-to-date. It is expected that build systems with more sophisticated signature processing and dependency analysis perform worse than those with more simple ones.

A developer will most probably only work on a small subset of the project at a time. As a consequence he will not need to rebuild the whole project to test the changes he made. So another key feature is to build a *specific target*.

As a typical developer's task, a small subset of source files is altered and a recompile is issued. It is expected that higher quality dependency analysis and a more fine-grained signature model will countervail the additional overhead. However, this is difficult to conduct as an automated test and thus not part of the benchmark.

Another interesting property measured is targeting the scheduling algorithm which walks the DAG. The test project is built using *parallel jobs* on multiple processors and multi-core platforms.

6.3.2 Scalability Test

As projects grow bigger and become more complex, so do the demands on the build system. It is hard to model the complexity and benchmark the build performance on real life projects. Hence, a synthetic project generator has been developed to observe the behavior when the number of targets, libraries, repositories is tweaked. Due to the synthetic nature of the benchmark, the results must be taken with a grain of salt. However, it will help to gain insight in how the processing and memory footprint scales with growing complexity of a software system.

6.3.3 Test Framework

6.3.3.1 Used Build Systems

The scalability test is carried out with the build systems used to build Bastei. To identify shortcomings in the specific way Make and SCons are used, minimal featured variants of Make and SCons build systems are also measured.

The minimal featured non-recursive Make variant is based on the design proposed by Miller [Mil97]. Implicit dependencies are extracted through the compiler as a side effect of compiling the source files. Target specifications are included in the top-level Makefile. The minimal featured SCons variant uses the SCons API directly. This should give an indication of the raw parsing and command execution capabilities.

Version 3.81 of GNU Make and SCons 0.98 were used in the benchmarks.

6.3.3.2 Build Platforms

The benchmarks are performed on Linux 2.6.22 on a variety of hardware platforms:

- AMD Athlon 64 Processor 3700+, 1GB main memory
- Intel Core Duo T2250 1.73GHz, 2GB main memory
- Intel Core 2 Quad-Core 2.4GHz, 4GB main memory

6.3.3.3 Measurement Tools

To measure runtime performance the GNU time utility is used.

Memory consumption is measured differently in Make and SCons. SCons reports the amount of virtual memory it uses when the `--debug=memory` flag is passed. The returned data was checked with Valgrind/Massif.

GNU Make does not support such a feature. Furthermore, if GNU Make executes in a recursive manner, it consists of a number of processes whose memory footprint must be accumulated. The memory usage of all running `make` processes is sampled and the maximum amount of memory used during the build process is recorded. This will ignore tools which strictly speaking belong to the build process like `sed` and `find`. Sampling artifacts also impose an error onto the measurement results.

Neither method is accurate. However, the results shall give an indication of the order of magnitude of memory demand. The memory consumption of the compiler or linker invocations is not included as it varies greatly depending on the input data.

6.3.4 Benchmark

6.3.4.1 Bastei build

Figure 6.1 on the next page illustrates the build performance of the Bastei project built for a Linux host platform. The most common use cases were tested. Surprisingly, SCons outperforms GNU Make in the full build and up-to-date check scenarios and is only minimally slower when building specific targets.

Why is the GNU Make-based Bastei build system such slow compared to the SCons based variant? In the previous chapters it was stated, that the SCons signature model was more computational challenging than the simple Make model. However, the recursive GNU Make configuration employed in the Bastei build system has to execute a lot of utilities. Make itself is spawned 293 times to perform a full build. The same Makefiles

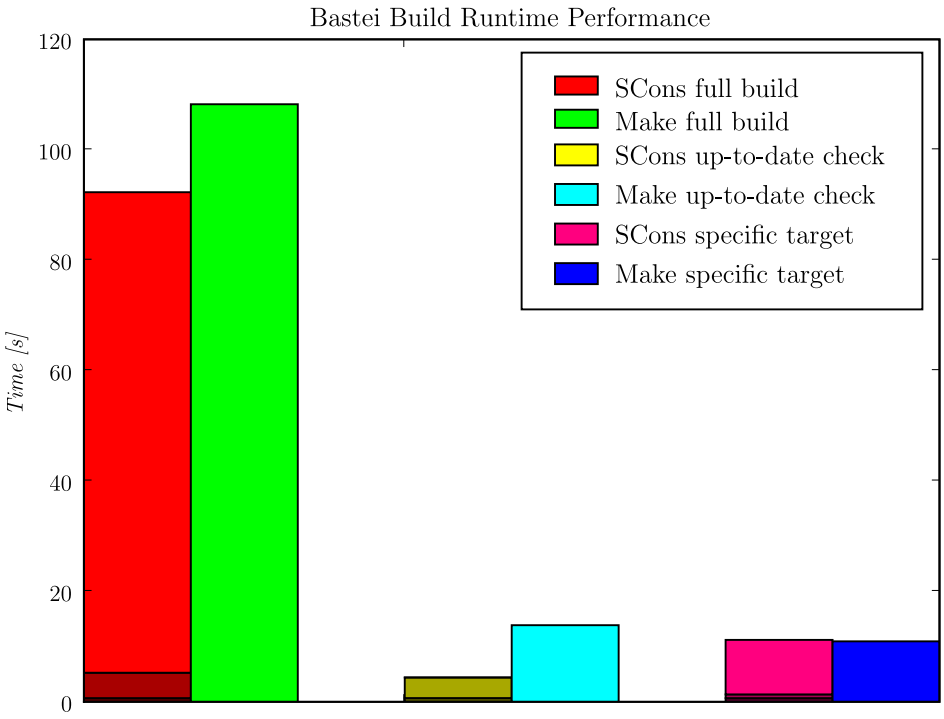

Figure 6.1: Bastei Build Performance on Intel Core Duo

are parsed redundantly for each target. Moreover, the process creation overhead must be considered.

The full build compiles 117 source files, converts 31 binary files, and links 27 targets and 18 libraries in the current configuration. There is a significant runtime difference between GNU Make and SCons performing the same build. The next section will evaluate the parallel build performance.

6.3.4.2 Parallel build

The build system itself takes a non-negligible share of the total build time. This is illustrated by the runtime of the up-to-date check in figure 6.1 and becomes important for the application of Amdahl's law. Typically, large parts of the build system execution itself is not parallelized and the build commands are run concurrently.

Figure 6.2 on page 31 depicts the behavior of Make and SCons on a quad-core processor with hyper-threading units. The biggest speedup can be achieved when switching from one job to two parallel jobs (see Table 6.2 on the next page).

Furthermore, SCons exploits the four cores much better than the recursive Make variant. A reason is that SCons keeps the whole DAG in memory and can schedule totally unrelated jobs in parallel. In contrast, recursive Make only processes one target

Table 6.2: Parallel build Bastei speedup

Jobs	1	2	3	4	5	6	7	8
GNU Make	1.00	1.50	1.69	1.76	1.77	1.79	1.79	1.80
SCons	1.00	1.92	2.73	3.37	3.51	3.67	3.83	3.91

at a time and can only schedule compilation steps for the target it processes. This means that Make runs out of potential jobs just before the target is finished. In the degenerated case, that there are a large number of single source targets, Make would only be able to schedule one job at a time, regardless of the number of available processors. However, it must be emphasized that Make is able to exploit parallelism as long as it isn't used recursively; or the recursive work sets are big enough.

Another observation is that additional virtual processors, whose number exceed the actually available processors, lead to better build performance. However, there is a limit beyond which the speedup declines again. An empiric rule of thumb indicates that twice the number of virtual processors is a good estimate for optimal build performance in most use cases.

Amdahl's law states that the achievable speedup is determined by the fraction of the system that must be executed serially [Amd67]. In the case of build systems, most of the work is done serially while external commands are executed in parallel. When ignoring precedence and resource constraints, the serially executed build system overhead solely determines the achievable build performance. Therefore, the higher the computational overhead of the build system, the lower the achievable speedup.

For reference, the build systems' parallel build performance is illustrated in figure 6.3 on page 32 for different build configurations of a synthetically generated project. The results match the initial assumptions of the behavior of the build systems. GNU Make is the fastest and reaches a speedup of 3.73 with 8 parallel jobs (see table 6.3). This shows that the parallel performance is primarily determined by the build configuration. The actual build system is of secondary importance.

Table 6.3: Parallel build speedup for synthetic project

Jobs	1	2	3	4	5	6	7	8
SCons* full build	1.00	1.66	2.17	2.30	**2.36**	2.35	2.35	2.35
SCons* specific target	1.00	1.09	1.13	1.13	1.14	1.13	**1.14**	1.13
SCons full build	1.00	1.74	2.26	2.41	2.46	**2.47**	2.46	2.45
SCons specific target	1.00	1.12	1.17	1.18	**1.19**	1.18	1.18	1.19
Make* full build	1.00	1.81	2.12	2.27	2.31	**2.32**	2.32	2.31
Make* specific target	1.00	1.78	2.08	2.25	**2.28**	2.19	2.19	2.19
Make full build	1.00	1.92	2.78	3.36	3.56	3.64	3.67	**3.73**
Make specific target	1.00	1.17	1.23	1.26	1.27	1.28	**1.28**	1.27

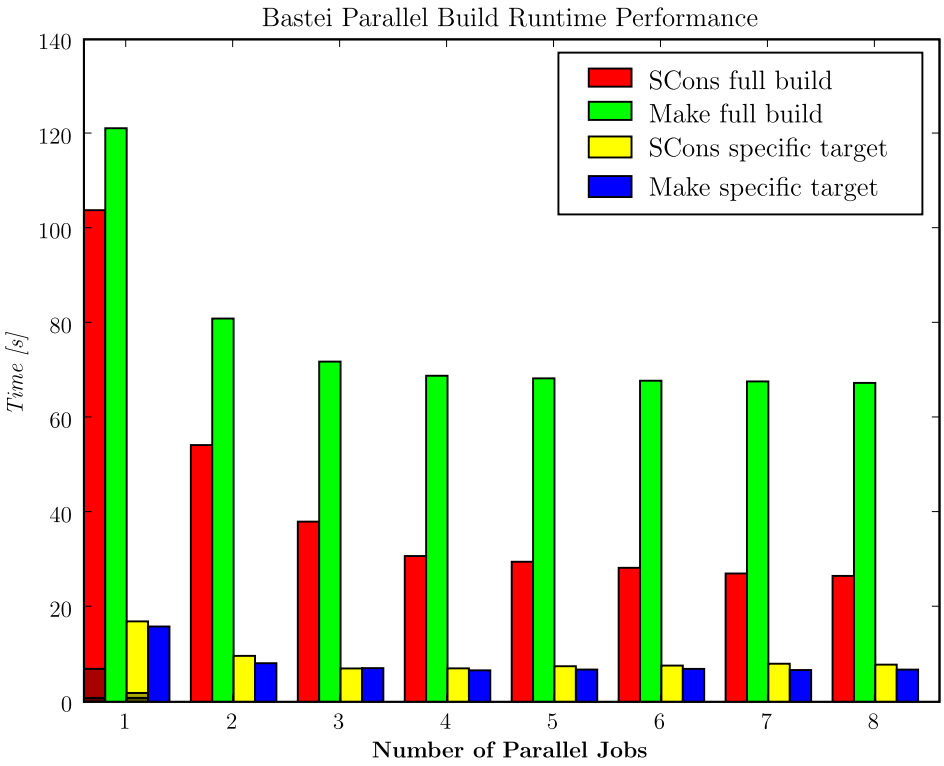

Figure 6.2: Parallel Build Performance for Bastei

6.3.4.3 Scalability test

Figure 6.4 on page 33 shows the runtime differences of full synthetic project builds. The Bastei build systems using Make and SCons were opposed to minimal featured variants. The runtime grows linearly with the number of nodes in the DAG.[1] Non-recursive Make is the fastest among the four, almost solely determined by the compiler and linker invocations. Bastei's Make build system has a huge overhead, it needs almost twice as much time compared to its non-recursive competitor.

The large differences must be partly ascribed to the characteristic of the synthetic projects: A large number of cheap-to-build files need to be generated. Therefore, the build system's performance is overstated in comparison to real-life projects, where the build system itself takes up significantly smaller shares.[2]

In the illustrated case, 21 source files constitute a target. The Bastei build system spawns a `make` instance for each target.

[1] The number of nodes depend on the number of source files but the latter is believed to be more illustrative.

[2] SCons' introspection reveals the time spent with parsing, build system execution and external command execution; illustrated by the shaded areas of the stacked bars.

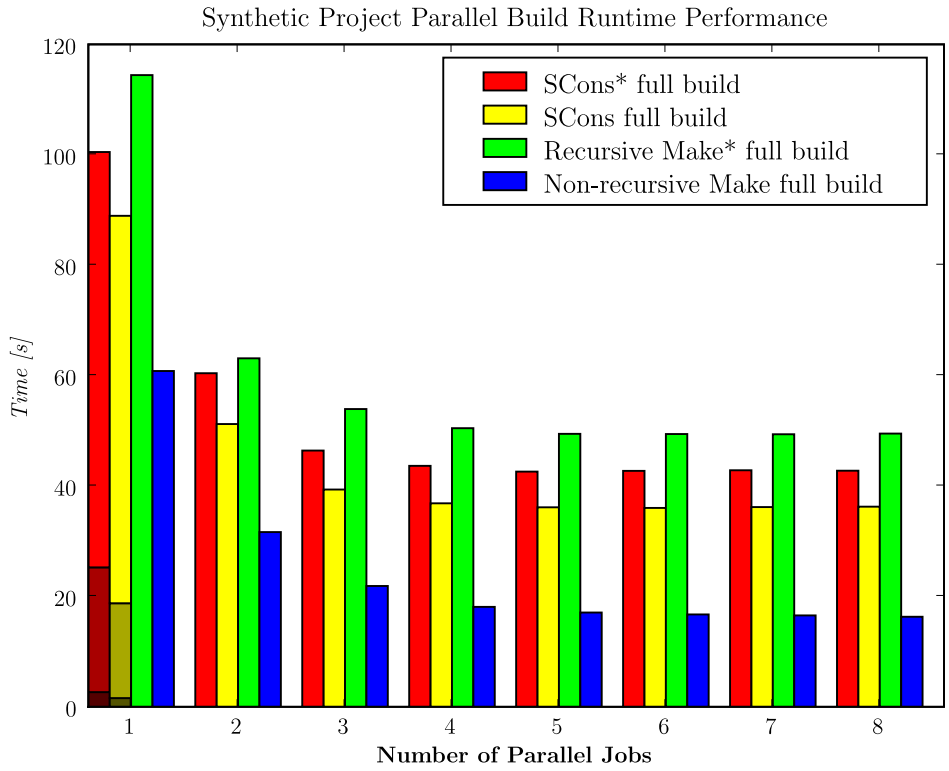

Figure 6.3: Parallel Build Performance for synthetic project

As can be seen in figure 6.5 on page 34, SCons needs a lot more memory than Make for performing the same task. There is a linear relationship between the number of source files and the amount of memory allocated in all build systems but recursive Make. The reason for the low memory footprint in recursive Make accounts to the layout of the synthetic project, which produces a large set of independent targets. Recursive Make spawns a separate process for each target and only needs to keep information about source files related to the current target. Therefore, only a small part of the DAG is kept in memory at a time.

Interestingly, there is a very big margin, almost one order of magnitude, between SCons and Make. The offset memory consumption of approximately 10MB accounts for the SCons source which must be loaded in advance. However, the fast linear growth in memory consumption is due to a large amount of information attached to each Node object. By extrapolating this graph, one can predict that a SCons based build system will be exposed to thrashing when the amount of physical memory is lower than what SCons requires. Another show-stopping effect happens when the available virtual memory is exhausted and SCons has to abort with a `MemoryError`.

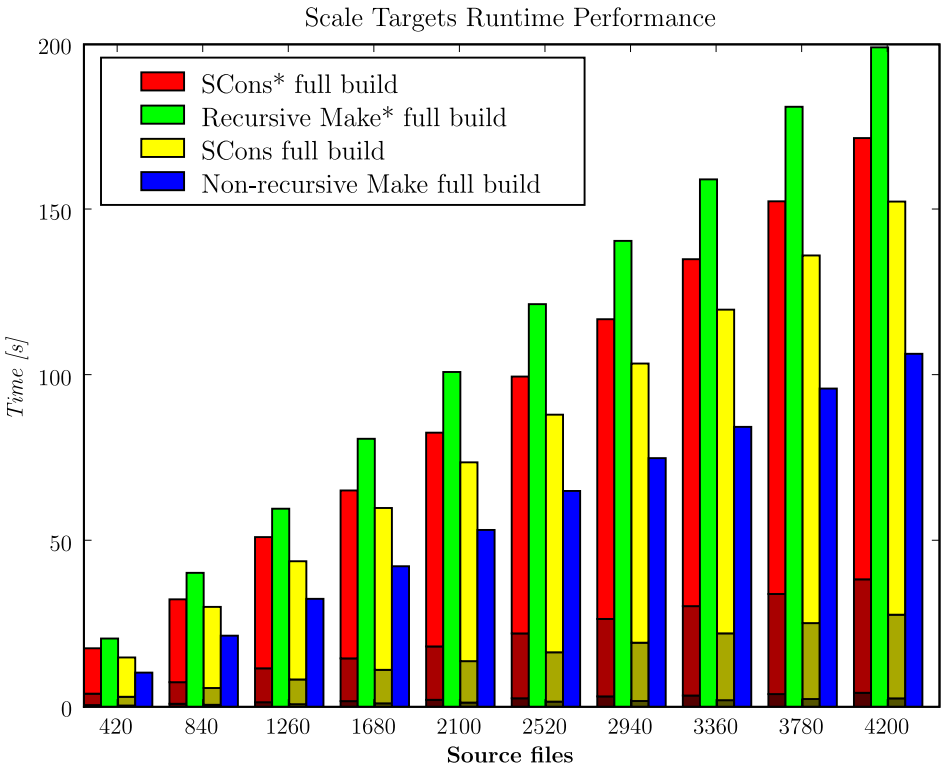

Figure 6.4: Scalability test runtime performance

6.4 Summary

Make and SCons have been evaluated from a feature and a performance perspective. SCons proved to be more accurate. Wrong build system decisions in Make are related to the stateless timestamp based file signature model. Build environment changes are not accounted for, files restored from archives are not handled correctly, and implicit dependency order may be violated. SCons on the other hand, has limited capabilities to extract implicit dependencies. If it has to resort to external tools to extract those, unnecessary recompilations are issued.

With regard to runtime performance and memory consumption, non-recursive Make proved to be most resource friendly. SCons has a decent runtime performance and is significantly faster than the tested recursive Make build system. However, SCons has an excessive memory footprint which scales linearly with the number of nodes. Recursive Make has the least memory consumption but the worst runtime performance. Another observation is that Make performance very much depends on the way it is used. While SCons offers a number of runtime introspection tools, Make users need to resort to profiling tools of the underlying system.

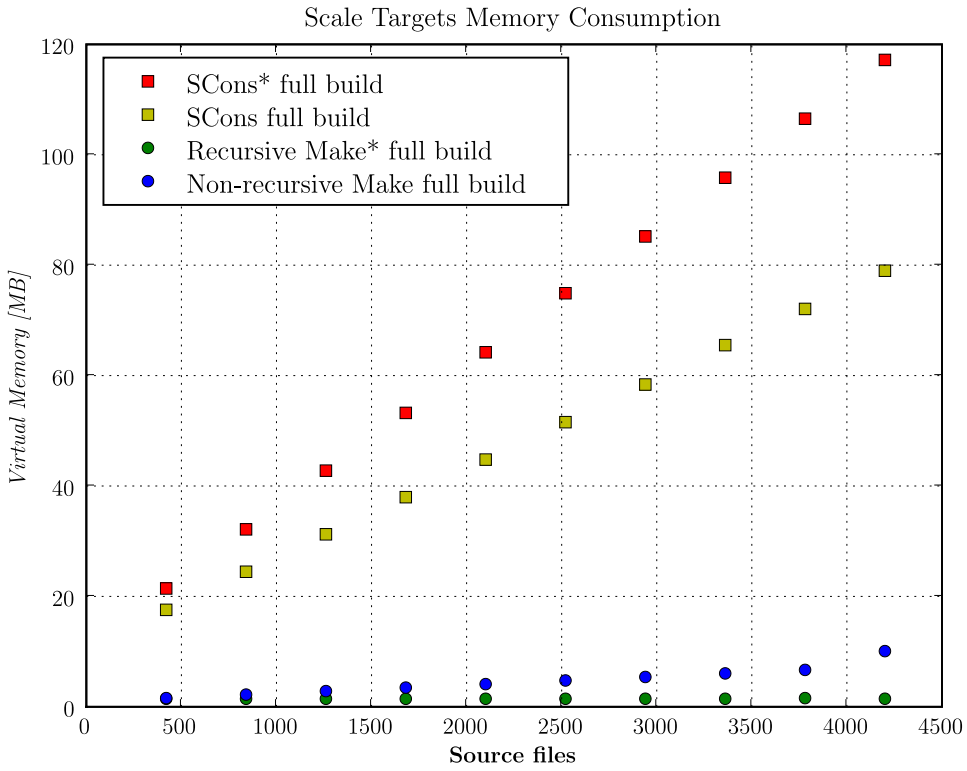

Figure 6.5: Scalability test memory consumption

SCons always builds the whole DAG, computes implicit dependencies, before any command is executed. That means there is a fixed amount of computational overhead, regardless how many commands are finally executed. Make, for example, can be used in a way that it will extract implicit dependencies on compilation of the target and store this information in a distributed cache. However, this may only speed up a full initial build.

Moreover, it is possible to reduce execution time and memory consumption by a great margin, by constructing only a partition of the dependency graph. This can be done if only a subset of targets need to be built and it is known in advance which build instructions need to be parsed. The Bastei build system employs this scheme.

7 Conclusion

Both contenders have a number of strengths and weaknesses. GNU Make and SCons are both general purpose build systems. Therefore, more specialized build systems may be more sophisticated in its respective domains.

The choice of Python as the build description language of SCons makes it possible to achieve very complex tasks without sacrificing readability. The object oriented approach makes it possible to address very advanced requirements in isolation of each other. However, there is a stiff learning curve if the user does not know Python in advance. The build descriptions tend to be a little more verbose compared to those of Make. That's mostly due to Python being a general purpose language, while Make's syntax was crafted for the problem domain.

SCons proved to be more accurate, mostly due to its stateful, content-based signature model.

On the other hand, GNU Make proved to be more resource friendly, especially regarding the memory footprint. SCons needs to address this problem to be a viable alternative to Make when building large software projects.

The benchmarks have revealed significant performance differences between different build systems, configurations and use cases. This indicates that it is worth to invest some time to analyze and optimize the build system of large software to save the developer's time. This requires tools which can give insight in the build system internals. Make lacks these tools and Makefile generators make it even more complicated to fine tune the build system. On the other hand, SCons provides a number of tools to analyze runtime performance.

The benchmark results back Miller's argument that recursive Make is indeed harmful [Mil97]. Not only is it more complicated to model interdependencies between separate modules because of which the developer has to make sure these precedences are honored. The benchmark results simply do not justify its usage. Especially the poor parallel performance exposes the conceptual weakness of the approach.

7.1 Future Work

SCons has demonstrated that content-based signatures are superior to simple timestamp-based signatures. One pitfall is the computational cost to generate those. There are a number of ways to avoid the expensive computation of content signatures. An idea is to monitor file system changes and only recompute the content hash if a write access was reported. Another approach could benefit from internal hashing some file systems exert (e.g. BTRFS). If a file's hash could be accessed by the build system, it would not need to compute checksums at all.

Another performance bottleneck is setting up the DAG each time a build is triggered, especially in large systems. The DAG could be made persistent and streamed in during successive builds. Some care need to be taken, to ensure consistency between the DAG and the file system, though.

The evaluated build systems treat each command equally when executing parallel jobs. This can lead to suboptimal schedules if the commands' runtimes have a high divergence. For example, in a build system which transcodes a movie next to performing traditional compilation tasks, the optimal schedule with regards to runtime would assign the transcoding to one processor and execute the compilation on the other one (on a dual processor platform). Transformation costs could be modeled by assigning weights to the edges of the DAG. These could be manually assigned, heuristically estimated (e.g. input complexity) or learned during the initial build [Baa88]. A more elaborate scheduling algorithm like [SRK07] could also help to better exploit multi-processor platforms.

It has been proposed to make use of build systems for goal oriented programming [vR98]. SCons already supports generic code execution for the Python language through Builders and can be used as the execution engine. However, it still lacks a facility to express constraints other than precedence and mutual exclusion. A semaphore mechanism would make the execution engine more powerful.

Glossary

DAG Directed acyclic graph

MD5 Message-Digest algorithm 5: cryptographic hash function with 128-bit hash value

Bibliography

[Amd67] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967. 8, 30

[Ant06] Apache Ant 1.7.0. Website, 2006. Available online at `http://ant.apache.org`; Visited on May 15th 2008. 9

[ATW94] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, 1994. URL: `http://grosskurth.ca/bib/entries.html#1994/adams`. 5

[Baa88] Erik. H. Baalbergen. Design and Implementation of Parallel Make. *Computing Systems*, 1(2):135–158, 1988. 6, 11, 17, 36

[Bor89] Ellen Borison. *Program changes and the cost of selective recompilation.* PhD thesis, Carnegie Mellon University, 1989. Note: Technical Report CMU–CS–89–205. 5

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, second edition, 2001. 6

[Fel79] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice & Experience*, 9(4):255–265, 1979. 11

[FH06] Norman Feske and Christian Helmuth. Design of the Bastei OS Architecture. Technical report, Institute for System Architecture, Operating Systems Group, December 2006. 21

[GGJ78] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance Guarantees for Scheduling Algorithms. *Operations Research*, 26(1):3–21, 1978. 6

[Gro07] Alan Grosskurth. Purely top-down software rebuilding. Master's thesis, University Of Waterloo, 2007. 11

[Kit08] Kitware, Inc. *CMake 2.6 Documentation*, 2008. Available online at `http://www.cmake.org/HTML/cmake-2.6.html`; Visited on May 3rd 2008. 14

[Lan05] Stefan Lang. Rant – Flexible, Ruby based make. Website, 2005. Available online at `http://rant.rubyforge.org/`; Visited on May 15th 2008. 9

[Mav08] Apache Maven 2.0.9. Website, 2008. Available online at `http://maven.apache.org`; Visited on May 15th 2008. 9

[McC03] Andrew McCall. Stop the autoconf insanity! Why we need a new build system. Website, 2003. Available online at `http://freshmeat.net/articles/view/889/`; Visited on May 15th 2008. 1

[MED02] David MacKenzie, Ben Elliston, and Akim Demaille. *Autoconf - Creating Automatic Configuration Scripts*. Free Software Foundation, Boston, 2002. URL: `http://www.gnu.org/software/autoconf/manual/autoconf.pdf`. Note: Last updated 2 December 2002 for version 2.57. 13

[Mil97] P. Miller. Recursive make considered harmful, 1997. 3, 11, 28, 35

[MOTV08] Gordon Matzigkeit, Alexandre Oliva, Thomas Tanner, and Gary V. Vaughan. *GNU Libtool*. Free Software Foundation, Boston, 2008. URL: `http://www.gnu.org/software/libtool/manual/libtool.pdf`. Note: Last updated 24 January 2008 for version 2.2. 13

[MTDL08] David MacKenzie, Tom Tromey, and Alexandre Duret-Lutz. *GNU Automake*. Free Software Foundation, Boston, 2008. URL: `http://www.gnu.org/software/automake/manual/automake.pdf`. Note: Last updated 21 January 2008 for version 1.10.1. 13

[Neu06] Alexander Neundorf. Why the KDE project switched to CMake – and how. Website, 2006. Available online at `http://lwn.net/Articles/188693/`; Visited on May 3rd 2008. 14

[Poo03] Martin Pool. distcc, a fast free distributed compiler. Website, December 2003. Available online at `http://ccache.samba.org/`; Visited on May 15th 2008. 8

[Sei94] Christopher Seiwald. Jam - Make Redux. Website, March 1994. Available online at `http://www.perforce.com/jam/doc/jam.paper.html`; Visited on May 15th 2008. 9

[Sid98] Bob Sidebotham. Software construction with Cons. *The Perl Journal*, 3(1), 1998. URL: `http://www.foo.be/docs/tpj/issues/vol3_-1/tpj0301-0012.html`. 15

[SK88] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, 1988. 5

[SMS06] Richard Stallman, Roland McGrath, and Paul D. Smith. *The GNU Make Manual, edition: 0.70*. Free Software Foundation, Boston, 2006. URL: `http://www.gnu.org/software/make/manual/make.pdf`. Note: Last updated 1 April 2006 for GNU `make` version 3.81. 12

[SRK07] Nadathur Satish, Kaushik Ravindran, and Kurt Kreutzer. A Decomposition-based Constraint Optimization Approach for Statically Scheduling Task Graphs with Communication Delays to Multiprocessors. In *Proceedings of the conference on Design, automation and test in Europe*, 2007. 36

[Thi03] Erik Thiele. Compilercache 1.0.10. Website, 2003. Available online at `http://www.erikyyy.de/compilercache/`; Visited on May 15th 2008. 8

[Tic86] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, 1986. 5

[Tic88] Walter F. Tichy. Tichy's response to R. W. Schwanke and G. E. Kaiser's "Smarter recompilation". *ACM Transactions on Programming Languages and Systems*, 10(4):633–634, 1988. 5

[Tri04] Andrew Tridgell. ccache 2.4. Website, 2004. Available online at `http://ccache.samba.org/`; Visited on May 15th 2008. 8

[vR98] Robbert van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998. 36

[Wei04] Jim Weirich. Rake – Ruby Make. Website, 2004. Available online at `http://rake.rubyforge.org/`; Visited on May 15th 2008. 9

[Zad02] Erez Zadok. Overhauling Amd for the '00s: A Case Study of GNU Autotools. In *Proceedings of Usenix Annual Technical Conference*, 2002. 13