ElectricAccelerator
Electric Make User Guide

Version 10.1

Electric Cloud, Inc.
125 South Market Street, Suite 400
San Jose, CA 95113
www.electric-cloud.com

ElectricAccelerator version 10.1

Trademarks

Contents

Chapter 1: eMake Overview 1-1

Understanding Component Interactions 1-2

eMake and EFS 1-2

eMake and Cluster Manager 1-2

ElectricAccelerator Virtualization 1-3

Electric File System (EFS) 1-3

System Registry (Windows Only) 1-3

User Accounts 1-4

Environment Variables 1-4

Understanding Build Parts 1-5

Chapter 2: Setting Up ElectricAccelerator 2-1

Defining Your Build 2-2

Build Sources 2-2

Build Tools 2-2

Build Environment 2-2

Configuring Your Build 2-4

Chapter 3: eMake Basics 3-1

Invoking eMake 3-2

Single Make Invocation 3-2

Setting the eMake Root Directory 3-3

Configuring Tools 3-4

Tools that Access or Modify the System Registry 3-5

Configuring Environment Variables 3-5

Setting the Cluster Manager Host and Port 3-6

Setting eMake Emulation 3-7

eMake Command-Line Options, Environment Variables, and Configuration File 3-8

Editing the eMake Configuration File 3-9

List of Command-Line Options 3-9

ElectricAccelerator Sample Build 3-24

Chapter 4: Additional eMake Settings and Features 4-1
Using Build Classes 4-2
Using Priority Pools 4-5
Using the Proxy Command 4-8
Using Subbuilds 4-9
Subbuild Database Generation 4-10
Run a Build Using Subbuild 4-10
Subbuild Limitations 4-10
Building Multiple Targets Simultaneously 4-12
Using eMake Variables 4-12
Using the Ninja Build System 4-13
Specifying Pragmas in an Addendum File 4-13
Restrictions 4-14
Supported Pragmas 4-14
Examples 4-15
Specifying the File to Use 4-15
Stopping a Build 4-16
Shutting Down Cluster Hosts During Builds 4-17
Chapter 5: Make Compatibility 5-1
Unsupported GNU Make Options and Features 5-2
Unsupported GNU Make Options 5-2
GNU Make 3.81 Support 5-2
GNU Make 3.82 Support 5-2
GNU Make 4.0 and 4.1 Support 5-2
GNU Make 4.2 Support 5-3
Unsupported NMAKE Options 5-3
Commands that Read from the Console 5-3
Transactional Command Output 5-4
Stubbed Submake Output 5-5
Submake Stubs 5-6
Submake Stub Compatibility 5-8
Hidden Targets 5-11
Wildcard Sort Order 5-13
Delayed Existence Checks 5-13
Multiple Remakes (GNU Make only) 5-14
NMAKE Inline File Locations (Windows only) 5-15

How eMake Processes MAKEFLAGS 5-15

Chapter 6: Performance Optimization 6-1

Optimizing Android Build Performance 6-2

Dependency Optimization 6-2

Enabling Dependency Optimization 6-2

Dependency Optimization File Location and Naming 6-2

Job Caching 6-3

Benefits 6-4

Limitations 6-4

Supported Tools 6-4

Running a "Learning" Build to Populate the Cache 6-4

Extending JobCache to Teams Via a Shared Cache 6-5

Configuring JobCache 6-6

Job Caching for gcc, clang, Jack, and javac 6-6

Job Caching for cl 6-9

Troubleshooting 6-13

Viewing JobCache Metrics 6-15

Moving Your Workspace 6-16

Deleting the Cache 6-16

Job Caching for kati 6-16

Parse Avoidance 6-17

Enabling Parse Avoidance 6-17

Deleting the Cache 6-19

Moving Your Workspace 6-19

Limitations 6-20

Troubleshooting 6-20

Ignored Arguments and Environment Variables 6-21

Javadoc Caching 6-23

Enabling Javadoc Caching 6-23

Limitations 6-24

Schedule Optimization 6-24

How it Works 6-24

Using Schedule Optimization 6-24

Disabling Schedule Optimization 6-25

Running a Local Job on the Make Machine 6-25

Jobs That Are Suited to Running Locally 6-25

Specifying Jobs to Run Locally 6-25
Making eMake Detect Files Outside the Current Working Directory 6-26
Serializing All Make Instance Jobs 6-26
Splitting PDBs Using hashstr.exe 6-27
Managing Temporary Files 6-28
Configuring the eMake Temporary Directory 6-28
Deleting Temporary Files 6-29

Chapter 7: Dependency Management 7-1
ElectricAccelerator eDepend 7-2
Dependency Generation 7-2
The Problem 7-3
eDepend Benefits 7-3
How Does eDepend Work? 7-3
Enabling eDepend 7-4
Using #pragma noautodep 7-5
ElectricAccelerator Ledger File 7-6
The Problem 7-7
The eMake Solution 7-8
Managing the History Data File 7-9
Setting the History File Location 7-10
History File Input Rules 7-10
History File Output Rules 7-10
Guaranteeing Correct History 7-10
Ensuring that Relative EMAKE_ROOT Locations Match 7-11
Running Builds with Multiple Roots 7-11
Using the remaphist Utility to Relocate a History File 7-11
Conflicts and Conflict Detection 7-13
How eMake Guarantees Reliable Parallel Builds 7-13
The Versioned File System 7-13
Detecting Conflicts 7-14
Exceptions to Conflict Detection in eMake 7-15

Chapter 8: Annotation 8-1
Configuring eMake to Generate an Annotation File 8-1
Annotation File Splitting 8-2
Working with Annotation Files 8-3
Creating Tools for Tasks That Use Annotation Output 8-3

Annotation XML DTD 8-3
Metrics in Annotation Files 8-6
Timer Annotation 8-6
Other Annotation 8-11

Chapter 9: Third-Party Integrations 9-1
Using ClearCase with ElectricAccelerator 9-2
Configuring ElectricAccelerator for ClearCase 9-2
ecclearcase Executable 9-2
LD_LIBRARY_PATH 9-2
ClearCase Views on Agents 9-2
--emake-clearcase 9-3
eMake’s “Fake” Interface for ClearCase 9-3
Where ClearCase Dynamic Views Affect eMake Behavior 9-5
Performance Considerations 9-7
Using Coverity with ElectricAccelerator 9-7
Using Cygwin with ElectricAccelerator (Windows Only) 9-8
Using Eclipse with ElectricAccelerator 9-8

Chapter 10: Electrify 10-1
Limitations 10-1
Recommendations 10-1
Electrify as Part of the Build Process 10-1
Running Electrify on Windows 10-1
Running Electrify on Linux 10-1
Important Reminders About Electrify 10-2
Electrify Arguments 10-2
Using Whole Command-Line Matching and efpredict 10-4
Whole Command-Line Matching 10-4
efpredict 10-5
Important Notes 10-6
Additional Electrify Information 10-6

Chapter 11: Troubleshooting 11-1
Agent Errors Establishing the Virtual File System 11-2
Agents Do Not Recognize Changes on Agent Machines 11-2
eMake Debug Log Levels 11-2
Enabling eMake Debug Logging 11-2
eMake Debug Log Level Descriptions 11-2

Using the Annotation File to Troubleshoot Builds 11-5
Determining Why a Target Was Rebuilt 11-5
Profiling Metrics 11-6
Per-Job Performance Metrics 11-6
Index 12-1

Chapter 1: eMake Overview

Electric Make® (”eMake”) is a new Make version and is the main build application in ElectricAccelerator®. eMake reads makefiles in several different formats, including GNU Make and Microsoft NMAKE. eMake distributes commands to the cluster for remote execution and services file requests. You can invoke eMake interactively or through build scripts.

Topics:

- Understanding Component Interactions on page 1-2
- ElectricAccelerator Virtualization on page 1-3
- Understanding Build Parts on page 1-5

Understanding Component Interactions

To a user, ElectricAccelerator ("Accelerator") might appear identical to other Make versions—reading makefiles in several different formats and producing identical results. Using a cluster for builds is transparent to the Accelerator user.

Important differences in Accelerator build processing versus other distributed systems:

- Components work together to achieve faster, more efficient builds. Instead of running a sequential build on a single processor, Accelerator executes build steps in parallel on a cluster of hosts.
- For fault tolerance, job results are isolated until the job completes. If an Agent fails during a job, Accelerator discards any partial results it might have produced and reruns the job on a different Agent.
- Missing dependencies discovered at runtime are collected in a history file that updates each time a build is invoked. Accelerator uses this collected data to improve performance of subsequent builds.

eMake and EFS

High concurrency levels in Accelerator are enabled by the Electric File System (EFS). When a job such as a compilation runs on a host, it accesses files such as source files and headers through EFS. EFS records detailed file access data for the build and returns that data to eMake.

eMake acts as a file server for Agents, reading the correct file version from file systems on its machine and passing that information back to the Agents. Agents retain different file version information and do not rely on eMake's file sequencing ability to provide the correct version for a job. The Agent receives file data, downloads it into the kernel, notifying EFS, which then completes the original request. At the end of a job, Electric Agent returns any file modifications to eMake so it can apply changes to its local file systems.

eMake and Cluster Manager

When eMake is invoked on the build machine, it communicates with Cluster Manager to acquire a set of Agents it can use for the build. When eMake finishes, it sends Cluster Manager the build results, and tells Cluster Manager that Agents are free now to work on other builds. If more than one build is invoked, Cluster Manager allocates agents using a priority-based algorithm. Builds with the same priority share Agents evenly, while higher priority builds are allocated more Agents than lower priority builds. By default, agents running on the same host machine are allocated to the same build. In real time, Cluster Manager dynamically adjusts the number of agents assigned to running builds as each build's needs change, which allows Accelerator to make the best use of cluster resources.

ElectricAccelerator Virtualization

ElectricAccelerator is designed to virtualize parts of the build setup so cluster hosts can be configured correctly for the specific build they are executing. Specifically, ElectricAccelerator dynamically mirrors the following host system properties on the Agent:

- Electric File System
- System registry (Windows only)
- User ID (UNIX only)
- Environment variables

Electric File System (EFS)

Files are the most important resource to distribute to hosts. When eMake starts, it is given a directory (or list of directories) called `EMAKE_ROOT`. All files in eMake root(s) are automatically mirrored on cluster hosts for the build duration. This powerful feature means you do not need to configure file sharing between the build system and the cluster—simply specify the `EMAKE_ROOT` and the hosts can access files on the host build system.

Almost any file visible to the host build system can be mirrored to the cluster—regardless of how the host system accesses that file (local disk, network mount, and so on). Both sources and tools can be mirrored, but there are important flexibility/performance "trade-offs" to consider when you include tools in the eMake root. See Configuring Your Build on page 2-4.

When the build completes, EFS is unmounted and the files are no longer visible on the hosts—this ensures builds do not interfere with local host configuration.

Generally, `EMAKE_ROOT` can be set to include any existing directory on the host build machine. Files in these directories are automatically accessible by commands running on cluster hosts. The exceptions are detailed in Setting the eMake Root Directory on page 3-3.

System Registry (Windows Only)

A side effect of running builds on Windows: Windows might execute tools that modify the system registry. For example, a build job step might install a program that includes registry modifications that are executed by a subsequent step. If job steps are run on different Agents without registry virtualization, the build might fail because registry modifications on one machine are not visible on another.

ElectricAccelerator solves this situation by mirroring the Windows registry in addition to the file system. You can specify a registry root using the command line `--emake-reg-roots=<path>`. Just as `EMAKE_ROOT` specifies a host file system subset to mirror, the registry root specifies which registry keys should be virtualized to Agents (for example, `HKEY_LOCAL_MACHINE\Software\My Product`). Access and modifications to keys and values under the registry root on Agents are then sent back to the host build system to ensure correct values are propagated to other Agents.

As with the file system, registry mirroring is active during the build only. After the build completes, the registry returns to its original configuration. Processes not running under an Agent on a host cannot see mirrored files and registry entries—they will see only the usual view of the machine.

User Accounts

The same user ID is used by the Electric Agent [running on each host, executing processes] and the eMake process started on the host build system. Using the same user ID ensures processes have the same permission and obtain the same results from system calls (such as `getuid`) as they would running serially on the user machine. On Windows, processes are executed by the Electric Agent service user. The Agent user can be configured at installation time, see the *ElectricAccelerator Installation Guide* at http://docs.electric-cloud.com/accelerator_doc/AcceleratorIndex.html.

Environment Variables

When eMake sends commands for execution to the Agent, it also sends environment variables. In most cases, this automatic environment virtualization is sufficient. Typically, variables that specify output locations, target architecture, debug-vs.-optimized, and so on, are propagated to tools on the host and their remote behavior correctly matches what they do locally. By default, environment variables sent by eMake override system-wide settings on the host.

For certain variables, override behavior is not desirable. For the build to operate correctly, tools running on the hosts must see the value from the local Agent, not the host build system. For example, the Windows `SYSTEMROOT` variable depends on the directory where Windows was installed on that machine—values might vary in different Window releases. The `SYSTEMROOT` value from the host build system is frequently different than the value from the local Agent.

To accommodate these instances, eMake allows you to exclude specific environment variables from virtualization. In addition to variables you might explicitly choose to exclude, eMake automatically excludes the `TMP`, `TEMP`, and `TMPDIR` variables on all platforms; and the `COMSPEC` and `SYSTEMROOT` variables on Windows. For a complete description of environment variables, see eMake Command-Line Options, Environment Variables, and Configuration File on page 3-8

Understanding Build Parts

Software builds are often complex systems with many inputs that must be carefully set up and configured. Correctly distributing system processes over a cluster of computers or hosts requires efficient replication of relevant parts of the build setup on each host.

ElectricAccelerator software is designed to help virtualize your build environment on each cluster host so distributed jobs display the same behavior as compared to how those jobs run serially on one computer.

Because input configurations can vary from one build to the next, ElectricAccelerator virtualization is active only while the build is running. When the build completes, the host returns to its original state. Consequently, different builds can share a cluster without danger of conflicting changes corrupting host configuration.

Chapter 2: Setting Up ElectricAccelerator

After all components are installed, and before invoking eMake, you need to configure ElectricAccelerator to ensure accurate, reliable builds.

Topics:

- Defining Your Build on page 2-2
- Configuring Your Build on page 2-4

Defining Your Build

Before using Accelerator, precisely define which components go into your build. Generally, a build has three input types:

SOURCES + TOOLS + ENVIRONMENT = BUILD

Build Sources

Build sources include all compiled or packaged files to build your product. For example, these sources include:

- Intermediate files generated during the build (for example, headers or IDL files)
- Makefiles and scripts that control the build
- Third-party source files read during the build
- Any file read by the build in source control

Build Tools

Tools used to create build output include make, compilers, linkers, and anything else operating on the sources during the build:

- Executable post-process tools (for example, *strip*)
- Static analyzers (for example, *lint*) if they run as part of the build
- Packaging tools (for example, *tar* or *zip*)

Sometimes the distinction between sources and tools is blurred. Consider these examples:

- A utility executable compiled from your sources during the build, run during later steps, but not part of final output
- Header source files that are part of the compiler (for example, `stdio.h)` or a third-party package, but not under source control

In this context, *sources* are those files that might change from one build to the next. Thus, a utility executable compiled from your sources as part of the build is considered a source.

By contrast, *tools* change infrequently—tools are often configured once and served from a central location (for example, an NFS share). A standard header such as `stdio.h` is usually considered part of the tool suite.

The distinction between inputs that can change between builds (sources) and inputs that can safely be assumed to be constant (tools) becomes important when configuring Accelerator virtualization.

Build Environment

Your operating system environment is an essential part of your build. The operating system is easy to overlook because the environment usually is configured once per host and then ignored as long as builds function normally. But when a build is distributed across a shared cluster, it is important to identify which parts of the operating system could affect the build. Some inputs to consider include:

- User environment variables
- System registry (Windows only)
- Operating system version (including patches or service packs)
- User account and user permissions
- Host-specific attributes (for example, machine name, network configuration)

For each of these inputs, consider what impact (if any) they will have on the build.

Generally, some environment variables require correct settings for a build (for example, `PATH` or variables that specify output architecture or source/output file locations).

Another common example of how the operating system can affect the build occurs with the use of tools that require license management. If a tool license (for example, a compiler) is host-locked or it requires contacting a license server to operate, ensure the compiler on the host can acquire the license also (for example, cluster host names could be configured as valid license server clients).

Note: Extended file attributes (xattr) are not supported. Attempting to query or set extended attributes for a file returns an `ENOTSUPP` (“Operation not supported”) error.

Configuring Your Build

After defining your build environment and identifying all of its inputs, configure the cluster and eMake so the system correctly virtualizes:

1. Set the eMake root directory.
2. Determine if you need to do additional configuration for tools.
3. Make additional configurations regarding the registry. (Windows only)
4. Configure environment variables if needed.
5. Set the Cluster Manager host and port.
6. Set eMake emulation.

Some cases of virtualization or distribution of specific job steps are not desirable. For these cases you can configure Accelerator to:

- Run an individual command from the Agent back on the host system, using the “proxy command” function. See Using the Proxy Command on page 4-8.
- Prevent remote job execution by using the `#pragma runlocal` function. See Running a Local Job on the Make Machine on page 6-25.

Chapter 3: eMake Basics

The following topics discuss basic eMake information to get you up and running.

Topics:

- Invoking eMake
- Setting the eMake Root Directory
- Configuring Tools
- Tools that Access or Modify the System Registry
- Configuring Environment Variables
- Setting the Cluster Manager Host and Port
- Setting eMake Emulation
- eMake Command-Line Options and Environment Variables

Invoking eMake

The eMake executable is called `emake`. The most important change to your build process is to ensure this executable is invoked in place of the existing Make.

For interactive command-line use, ensure the following:

- The ElectricAccelerator `bin` directory is in your `PATH` environment variable:
  - For Linux: `<install_dir>/i686_Linux/bin` (`/opt/ecloud/i686_Linux/bin` by default) or `<install_dir>/i686_Linux/64/bin` (`/opt/ecloud/i686_Linux/64/bin` by default)
  - For Solaris: `<install_dir>/sun4u_SunOS/bin` (`/opt/ecloud/sun4u_SunOS/bin` by default)
  - For Windows: `<install_dir>\i686_win32\bin` (`C:\ecloud\i686_win32\bin` by default) or `<install_dir>\i686_win32\64\bin` (`C:\ecloud\i686_win32\64\bin` by default)
- You type `emake` in place of gmake or nmake.

Note: The 64-bit version of eMake is recommended for builds with very large memory requirements.

You can rename the eMake executable to either gmake or nmake, because eMake checks the executable to determine which emulation to use. If the name of the submake is hard-coded in many places within your makefiles, a simple solution would be to rename gmake or nmake to *gmake.old* or *nmake.old*, and rename eMake to either *gmake* or *nmake* on cluster hosts only. In this way, you can maintain access to your existing make, but all submakes from an Accelerator build will correctly use eMake.

Note: Electric Cloud does ***not*** recommend running builds in `/tmp`.

To ensure eMake is called for recursive submake invocations in makefiles, use the `$(MAKE)` macro for specifying submakes instead of hard-coding references to the tool. For example, instead of using:

```
libs:
        make -C lib
```

use the following `$(MAKE)` macro:

```
libs:
        $(MAKE) -C lib
```

Single Make Invocation

It is important to keep the build in a single Make invocation. At many sites, Make is not directly invoked to do a build. Instead, a wrapper script or harness is used to invoke Make, and users (or other scripts) invoke this wrapper. The wrapper script might take its own arguments and might perform both special set up or tear down (checking out sources, setting environment variables, post-processing errors, and so on). Because eMake behaves almost exactly like native Make tools, usually it can directly replace the existing makefile in wrapper scripts.

Sometimes, however, the script might invoke more than one Make instance. For example, the script could iterate over project subdirectories or build different product variants. In this case, each of these builds becomes a separate Accelerator build, with its own build ID, Cluster Manager entry, history file, and so on.

It is much more efficient for Make instances that are logically part of one build to be grouped under the control of a single parent Make invocation. In this way, eMake can track dependencies between submakes, ensure maximal parallelization and file caching, and manage the build as a single, cohesive unit.

If your build script invokes more than one submake, consider reorganizing makefile targets so a single Make is invoked that in turn calls Make recursively for submakes.

If a lot of the setup for each instance occurs within the build script, another possible solution is to use a simple top level makefile to wrap the build script; for example,

```
all:
        my-build-harness ...
```

In this instance, `my-build-harness` runs on the Agent much like any other command and sends commands discovered by submake stubs back to the host build machine. This approach works only if each submake's output is not directly read by the script between Make invocations. Otherwise, it might be susceptible to submake stub output problems. See Submake Stubs on page 5-6 for more information.

Setting the eMake Root Directory

The `--emake-root` option (or the `EMAKE_ROOT` environment variable) specifies the EFS root directory [or directories] location. All files under the eMake root directory are automatically mirrored on each Agent.

eMake uses the current directory as the default if no other root directory is specified. You must specify the correct root directory (or directories) or the build might fail because eMake cannot find the necessary files to complete the build or resolve dependencies.

For best results and performance, be specific when setting the eMake root location. Be sure to include:

- All files created or modified by the build.
- All source files.
- The location where build output files will go during the build, for example, object files, linker output, and so on.
- Other files read by the build such as third-party tools and system headers, or other files not modified if you need to. Be aware, however, that including these files can affect performance. See Configuring Your Build on page 2-4.

  If necessary, specify more than one directory or subdirectory. Separate each location using standard `PATH` variable syntax (a colon for UNIX, a semicolon for Windows).

  UNIX example:

  ```
  --emake-root=/src/foo:/src/baz
  ```

In this example, you have streamlined the root path by excluding other `/src` subdirectories not necessary to the build.

Windows example:

```
--emake-root=C:\Build2;C:\Build4_test
```

Note: Any files used by the build, but not included under an eMake root directory, must be preloaded onto all hosts and identical to corresponding files on the system running eMake. If these files are not identical, eMake could find and use the wrong files for the build. This approach is appropriate for system compilers, libraries, header files, and other files that change infrequently and are used for all builds.

Generally, `EMAKE_ROOT` can be set to include any existing directory on the host build machine. Files in these directories are automatically accessible by commands running on cluster hosts. However, there are a few exceptions:

- `EMAKE_ROOT` cannot be set to the system root directory (for example, "`/`" on UNIX or `C:/` on Windows). It might be tempting to try this to specify "mirror everything," but in practice, this is not desirable because mirroring system directories such as `/var` or `/etc` on UNIX or `C:/Windows` on Windows can lead to unpredictable behavior. eMake will not allow you to specify the root directory as `EMAKE_ROOT`.
- `/tmp` and the Windows temp directory cannot be included in the eMake root.
- On Windows, another operating system restriction is imposed:
  `EMAKE_ROOT` is not a UNC path specification—it must be a drive letter specification or a path relative to a drive letter. It must also be a minimum of three characters.

Configuring Tools

Cluster hosts must be able to execute all tools (compilers, linkers, and so on) required during the build. Three execution setup choices:

1. **Tools are available on the cluster in the same locations as on the host build system**—this is the simplest setup and requires no special eMake or build system configuration.

   This is the most common setup if tools are provided from a network mount (for example, an NFS or SMB server). If tools are not supplied from a network share, but are installed on the local disk on each host, those tools must be updated whenever tools on the build system change.

2. **Tools are available on the cluster, but not necessarily in the same locations as the host build system**—this configuration is common in environments where users are allowed to install tools on their host system in non-standard locations.

   In this case, care must be taken to ensure build steps executing on the hosts find the tools. Usually, this means making the `PATH` environment variable include standard locations used on the cluster hosts—any other variables referencing tool locations need to be modified similarly.

   As in the first case, if tools are installed on the cluster hosts' local disk, they must be updated when tools are updated on the host system.

3. **Tools are not installed on the cluster**—in this case, when no tools are installed on the cluster, `EMAKE_ROOT` is configured to include the tools directory so the compiler and linker are sent to the Agents by eMake [in the same manner as the source files].

   This is the most flexible setup because it eliminates the need to install or update tools on the hosts. If tools change frequently or vary between branches, this setup can dramatically reduce setup cost. However, to gain this flexibility, you typically trade performance because network overhead is increased.

   Another factor to consider is the possibility that configuration information for a virtualized tool might not work on the host. For example, it is not possible to virtualize the Microsoft Visual Studio IDE because of its tight integration with the local machine's registry.

In some special cases, it might be necessary to use tools outside of the virtualized file system. To support running such tools on the local build machine during a cluster build, see Using the Proxy Command on page 4-8.

IMPORTANT: Electric Cloud does not recommend Proxy Command for frequent use; use it with caution.

Tools that Access or Modify the System Registry

In addition to files, tools on Windows might access or modify the system registry. Use the `--emake-reg-roots` command-line option to specify a key to mirror. You can specify more than one key by separating multiple entries with semicolons:

```
--emake-reg-roots=HKEY_LOCAL_MACHINE\Software\Foo;
HKEY_LOCAL_MACHINE\Software\Bar
```

In addition, you can specify exception keys to not mirror the system registry by prefixing the key with a "`-`" character. For example:

```
--emake-reg-roots=HKEY_LOCAL_MACHINE\Software\Foo;
-HKEY_LOCAL_MACHINE\Software\Foo\Base
```

which means "mirror all keys and values under `Foo` except the keys in `Foo\Base`.

To ensure compatibility with Microsoft Visual Studio, the registry root specification automatically includes:

```
HKEY_CLASSES_ROOT;-HKEY_CLASSES_ROOT\Installer;
-HKEY_CLASSES_ROOT\Licenses
```

Configuring Environment Variables

If your build environment is fairly constant, you might want to use *environment variables* to avoid respecifying values for each build. Environment variables function the same way as command-line options, but command-line options take precedence over environment variables.

Because eMake environment variables are propagated automatically to Agents, most commands running on an Agent will run with the correct environment without modifications.

However, two important exceptions exist.

- As described in Configuring Tools on page 3-4, if tools are installed on the cluster host in different locations from the build system, the `PATH` variable (and other variables that reference tool locations) on the build system must be modified to include the locations for the cluster tools.
- Generally, differences between the build system and cluster hosts might indicate it is undesirable to override environment variables with eMake values. In this case, use the `--emake-exclude-env` command-line option or the `EMAKE_EXCLUDE_ENV` environment variable.

  For example, consider a build system environment variable called `LICENSE_SERVER` that normally contains the license server name that the system should contact to obtain a tools license. If this variable is machine-specific, eMake overrides the correct machine-specific value on the cluster hosts with the value from the build system. To ensure eMake does not override `LICENSE_SERVER` with the value from the build system, use the option, `--emake-exclude-env`, when running eMake:

  ```
  --emake-exclude-env=LICENSE_SERVER
  ```

  You can specify more than one value by separating them with commas:

  ```
  --emake-exclude-env=LICENSE_SERVER,TOOLS_SERVER
  ```

  Some variables are almost always used to describe the local machine state, so eMake always excludes them from mirroring. These variables are:

  ```
  TMP
  TEMP
  TMPDIR
  ```

  For Windows, this list also includes:

  ```
  COMSPEC
  SYSTEMROOT
  ```

Configuring ccache

The only configuration required to use ccache with Accelerator is to set the `CCACHE_NOSTATS` environment variable. If you do not set this environment variable, the entire build becomes serialized because ccache continuously writes to a statistics file throughout the build. To learn more about ccache, refer to http://ccache.samba.org/.

Setting the Cluster Manager Host and Port

The `--emake-cm` option (or the `EMAKE_CM` environment variable) sets the Cluster Manager host name (either IP address or machine name) and must be specified. If the host name is not specified by either of those methods, eMake operates in local mode and performs like a traditional make program by running jobs on the local machine in serial order—without using the cluster hosts.

Cluster Build Example

The Cluster Manager host name is `linuxbuilder`.

- You can invoke a build against the cluster by running:

```
% emake --emake-cm=linuxbuilder
```

- Alternatively, use the `EMAKE_CM` environment variable:

```
% setenv EMAKE_CM linuxbuilder
% emake
```

Local Machine Build Example

- To force a local build (one *not* run on the cluster), leave the Cluster Manager host name undefined by running:

```
% emake --emake-cm=
```

- Or, leave the `EMAKE_CM` environment variable is undefined:

```
% emake
```

Setting eMake Emulation

eMake can emulate different make variants: GNU Make (gmake), gmake in a Cygwin environment, Microsoft NMAKE (nmake), and Symbian. It can also emulate Ninja. By default, `--emake-emulation=<mode>` is set to `gmake`, which supports a subset of GNU Make 3.81 (see Unsupported GNU Make Options and Features on page 5-2).

The following modes are available:

Mode	Linux support	Windows support	Solaris support
`gmake4.2`	Yes	Yes	Yes
`gmake4.1`	Yes	Yes	Yes
`gmake4.0`	Yes	Yes	Yes
`gmake3.82`	Yes	Yes	Yes
`gmake`	Yes	Yes	Yes
`gmake3.81`	Yes	Yes	Yes
`gmake3.80`	Yes	Yes	Yes
`gmake3.79.1`	Yes	Yes	Yes
`symbian`	No	Yes	No
`nmake`	No	Yes	No
`nmake8`	No	Yes	No
`nmake7`	No	Yes	No

Mode	Linux support	Windows support	Solaris support
`cygwin`	No	Yes	No
`ninja`	Yes	No	No
`ninja1.7.2`	Yes	No	No

Note: You can rename `emake.exe` to `nmake.exe`, `gmake.exe`, and so on to change the emulation type for all builds automatically. See the `--emake-emulation-table` option in the "eMake Command-Line Options, Environment Variables, and Configuration File on page 3-8

NMAKE Emulation Example

To use NMAKE, use the following option to set the emulation type:

```
--emake-emulation=nmake
```

Cygwin Emulation Example

To use Cygwin, use the following option to set the emulation type:

```
--emake-emulation=cygwin
```

Ninja Emulation Example

To use Ninja, use the following option to set the emulation type:

```
--emake-emulation=ninja
```

eMake Command-Line Options, Environment Variables, and Configuration File

You can configure eMake options from the command line for a specific build and can also use eMake environment variables or the `emake.conf` configuration file to make options persistent.

Following are caveats for using these methods for setting options:

- The environment variable `EMAKEFLAGS` can be used to set any command-line option. For example, this emake invocation:

  ```
  % emake --emake-cm=mycm --emake-root=/home/joe
  ```

  is equivalent to the following in `csh`:

  ```
  % setenv EMAKEFLAGS "--emake-cm=mycm --emake-root=/home/joe"
  % emake
  ```

  The Bash shell equivalent is:

  ```
  $ export "EMAKEFLAGS=--emake-cm=mycm --emake-root=/home/joe"
  $ emake
  ```

  The Windows command shell equivalent is:

```
C:\> set EMAKEFLAGS=--emake-cm=mycm --emake-root=C:\home\joe
C:\> emake
```

- The hierarchy or precedence for setting an eMake option is:
    - Command-line options
    - Options set using the `EMAKEFLAGS` environment variable
    - Options set using other Electric Cloud environment variables

  Command-line options as well as options in the `EMAKEFLAGS` environment variable override options in the `emake.conf` configuration file.

Editing the eMake Configuration File

The path to the eMake configuration file is `<InstallDir>/<arch>/conf/emake.conf` (on Linux and UNIX) or `<InstallDir>\<arch>\conf\emake.conf` (on Windows), where `<InstallDir>` is the top installation directory containing the eMake program, and `<arch>` is the architecture (such as `i686_Linux` or `sun4u_SunOS`).

If you run an eMake executable that is not within an ElectricAccelerator installation, it will not use the installed configuration file (but will still use environment variables and options entered from the command line).

The configuration file has the following formatting rules:

- Only eMake-specific command line arguments are allowed (not those specific to GNU Make, NMAKE, and so on).
- One option (a switch-argument pair) is allowed per line.
- All allowed switches must be separated from their arguments by `=` (an equal sign).
- All text between `=` and the end of the line (including spaces and terminal spaces) is part of the argument.
- File order determines the option order (for many kinds of options, the final occurrence overrides all earlier ones).
- `NUL` characters are not allowed.
- Empty lines are ignored.
- Lines beginning with `#` are comment lines and are ignored.

Configuration file syntax errors (but not necessarily semantic ones) are warnings and cause the configuration file to be skipped (as if it was not present, except for the warnings).

List of Command-Line Options

Command-line options are listed in alphabetical order except for platform-specific options that are listed *after* platform-independent options. Debug options are listed at the end of the table.

Note: The `--emake-volatile` command-line option is deprecated and no longer has any effect. If the option is specified, it is ignored.

Command-line Options	Environment Variables	Description
–	`EMAKE_BUILD_MODE`	Always set to local. Specifies that an individual emake invocation on the Agent does not enter stub mode, but instead behaves like a local (non-cluster) make. eMake automatically uses local mode when the `-n` switch is specified.
`--emake-android-root=`*`<path>`*	–	Specifies the location of your Android source files. For details about Android builds, see the *KBEA-00165 - Best Practices for Android Builds Using ElectricAccelerator 10.1* KB article.
`--emake-android-version=`*`<version>`*	–	Specifies your Android version. The available values are `7.0.0`, `8.0.0`, or `9.0.0`. For details about Android builds, see the *KBEA-00165 - Best Practices for Android Builds Using ElectricAccelerator 10.1* KB article.
`--emake-annodetail=`*`var1[,var2[,...]]`*	–	Specifies the level of detail to include in annotation output—a comma-separated list for any of the following values: `basic`: Basic annotation (enabled by default in cluster mode). If the JobCache feature is enabled, basic annotation includes information about cache hits and misses. `env`: Enhanced environment variables `file`: Files read or written `history`: Serialization details `lookup`: All file names accessed `md5`: MD5 checksums for reads/writes `registry`: Updates to registry `waiting`: Jobs that waited `none`: Disables all annotation. Note that this value disables basic annotation, even when the `--emake-annofile` option is used. This option overrides the build class annotation settings set on the Cluster Manager.

Command-line Options	Environment Variables	Description

`--emake-annofile=<file>`	–	Specifies the name of the XML-formatted log output file. By default, the annotation file, `emake.xml`, is created in the directory where eMake is run. If specified, implies at least "basic" annotation details.
		The following macros are available:
		`@ECLOUD_BUILD_ID@` expands into the unique eMake build ID.
		`@ECLOUD_BUILD_TAG@` expands into the build tag from the Cluster Manager (this is displayed in the Name column on the Build tab in the Cluster Manager UI).
		`@ECLOUD_BUILD_DATE@` expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD.
		`@ECLOUD_BUILD_TIME@` expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS.
		Example:
		`--emake-annofile=annofile-@ECLOUD_BUILD_ID@-@ECLOUD_BUILD_TAG@-@ECLOUD_BUILD_DATE@-@ECLOUD_BUILD_TIME@.xml`
		results in:
		`annofile-4-default_4_20090220184128-20090220-184128.xml`

`--emake-annoupload=<0|1>`	–	Enables (`1`) or disables (`0`=default) annotation file upload on the Cluster Manager. This option overrides the build class annotation upload setting set on the Cluster Manager.
`--emake-autodepend=<0|1>`	–	Enables (`1`) or disables (`0`=default) the eDepend feature.
`--emake-big-file-size=<N>`	–	Sets the minimum file size (in bytes) to send through Agent-to-Agent transfers for direct file sharing between hosts. Default=`10KB`.
`--emake-build-label=<label>`	`ECLOUD_BUILD_LABEL`	Sets a customized build label. These labels are literal strings and do not use available tags when defining labels for build classes.

Command-line Options	Environment Variables	Description
`--emake-class=<class>`	`ECLOUD_BUILD_CLASS`	Specifies the build class for the current build. The class must match an existing Cluster Manager class previously created by a user. If this option is not used, or if the class does not match, eMake assigns the default class to the build.
`--emake-clearcase=var1[,var2[,...]]`	`EMAKE_CLEARCASE`	Turns on support for specified ClearCase features—a comma-separated list of any of the following values: `symlink` : symbolic links `vobs` : per-VOB caching (for speed) `rofs` : read-only file system
`--emake-cluster-timeout=<N>`	–	If no Agents are available when the build starts, this option specifies the number of seconds to try to acquire Agents before giving up. The default is `-1` (infinite), so the build waits indefinitely for Agents.
`--emake-cm=<host>[:port]`	`EMAKE_CM`	Sets the host name (either the IP address or machine name) and port for Cluster Manager. Note that you cannot use `--emake-cm` and `--emake-localagents=y` in the same eMake invocation.
`--emake-collapse=<0|1>`	–	Turns history collapsing on or off. When collapsing is enabled, dependencies between a single or several jobs in another *make* instance are replaced with a serialization between the job and the other Make instance. This action typically results in significant history file size reduction, but might cause some over-serialization. In most builds, this has little or no impact on build time. In some builds, disabling collapsing improves performance at the cost of increased history file size. Default=1 (on)
`--emake-disable-pragma=var1[,var2[,...]]`	–	Comma-separated list of pragma directives to ignore—can be one or more of: `allserial`, `runlocal`, `noautodep`, or `all` to disable all pragmas.
`--emake-disable-variable-pruning=<0|1>`	–	Disables variable table pruning. Default=`0` (off)

Command-line Options	Environment Variables	Description
`--emake-emulation=<mode>`	`EMAKE_EMULATION`	Sets Make-type emulation to a specific mode. Default emulation type is `gmake`. You can rename `emake.exe` to `nmake.exe` or `gmake.exe` to change the emulation type for all builds automatically. Available modes: `gmake`, `gmake3.80`, `gmake3.81`, `gmake3.82`, `symbian`, `nmake`, `nmake7`, `nmake8`, `cygwin`, `ninja` (Linux only), or `ninja1.7.2` (Linux only) If specifying `gmake3.82`, read the GNU Make 3.82 Support on page 5-2.
`--emake-emulation-table=<table>`	–	Configures default emulation modes for Make programs. TABLE is a comma-separated list of NAME=MODE, where NAME is the name of a Make executable and MODE is the emulation mode to use if `emake` is invoked as NAME.
`--emake-exclude-env=var1[,var2[,...]]`	`EMAKE_EXCLUDE_ENV`	Specifies which environment variables must not be replicated to the hosts.
`--emake-hide-warning=<list>`	`EMAKE_HIDE_WARNING`	Hides one or more Accelerator-generated warning numbers. *`list`* is a comma-separated list of numbers that you want to hide. The `EC` prefix of a warning number is optional and can be omitted.
`--emake-history=<read|create|merge>`	–	Specifies the history mode creation model. Default=`merge`.
`--emake-history-force=<0|1>`	–	Honors history mode even if the build fails. Default=`1` (on)

Command-line Options	Environment Variables	Description

`--emake-historyfile=` *`<path/file>`*	–	Specifies which history file to use for a specific build. Allows you to change the default name and path for the history file `emake.data` set automatically by eMake.
		The following macros are available:
		`@ECLOUD_BUILD_ID@` expands into the unique eMake build ID.
		`@ECLOUD_BUILD_TAG@` expands into the build tag from the Cluster Manager (this is displayed in the Name column on the Build tab in the Cluster Manager UI).
		`@ECLOUD_BUILD_DATE@` expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD.
		`@ECLOUD_BUILD_TIME@` expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS.
		Example:
		`--emake-historyfile=historyfile-@ECLOUD_BUILD_ID@-@ECLOUD_BUILD_TAG@-@ECLOUD_BUILD_DATE@-@ECLOUD_BUILD_TIME@.xml`
		results in:
		`historyfile-4-default_4_20090220184128-20090220-184128.xml`

`--emake-idle-time=` *`<N>`*	–	Sets the number of seconds (`N`) before idle Agents are released to the cluster. Default=`10`
`--emake-ignore-all-intermediate=<0|1>`	–	Causes eMake to not treat .SECONDARY targets having no prerequisites as meaning that all targets are intermediate. This option might increase performance on certain large builds with thousands of targets and reduce eMake runtime memory requirements (but breaks strict compatibility with GNU Make when emulating GNU Make 3.81 and later).
`--emake-impersonate-user=`*`<name>`*	`ECLOUD_IMPERSONATE_USER`	Run the build as this Cluster Manager user. **Note:** Impersonate changes the user recorded by the Cluster Manager and that user's permissions. This option does not affect OS user permissions. For additional information, see the online help topic, "Permissions."
`--emake-job-limit=` *`<N>`*	–	Limits the maximum number of uncommitted jobs to `N` where `0` means unlimited. Default=`0`

Command-line Options	Environment Variables	Description
`--emake-ledger=<valuelist>`	`EMAKE_LEDGER`	Enables the ledger capability. `Valuelist` is a comma-separated list that includes one or more of: `timestamp`, `size`, `command`, `nobackup`, `nonlocal`, and `unknown`.
`--emake-ledgerfile=<path/file>`	`EMAKE_LEDGERFILE`	The name of the ledger file. Default=`emake.ledger`
`--emake-maxagents=<N>`	–	Limits the number of Agents used by this build. `N=0`uses all available Agents. Default=`0` This option overrides the build class Max Agents setting if `--emake-maxagents` is set lower than the build class setting. This option does ***not*** override the build class setting if `--emake-maxagents` is set higher than the build class setting. This behavior prevents a user from overriding the upper limit for a class that was set up by the build administrator.
`--emake-mem-limit=<N>`	–	Controls the amount of memory eMake devotes to uncommitted jobs. When the limit is exceeded, eMake stops parsing new Make instances. Default=`1,000,000,000` (1 GB).
`--emake-mergestreams=<0/1>`	`EMAKE_MERGE_STREAMS`	Indicates whether to merge the `stdout/stderr` output streams, yes (`1`) or no (`0`). The default is merge the streams (`1`). For most situations, this is the correct value. If you re-direct standard output and standard error separately, specify no (`0`) for this option.
`--emake-monitor <hostname/IP>:<port>`	–	Sets the hostname/IP and port of the system from where you want to view the live monitor data in the Electric Cloud ElectricInsight® tool. To monitor live build data, you must launch the ElectricInsight live monitor *before* you start the build.
`--emake-pedantic=<0|1>`	–	Turns *pedantic mode* on (`1`) or off (`0`=default). When pedantic mode is on, warnings appear when invalid switches are used, or potential problems are identified (for example, rules with no targets or reading from a variable that was not written). When pedantic mode is off, eMake ignores irrelevant switches or exits without warning if it encounters unresolvable errors.

Command-line Options	Environment Variables	Description
`--emake-pragmafile=<file>`	–	Specifies the pragma addendum file to use. This is a file containing additional pragma declarations to apply to the build. eMake with Ninja emulation supports pragmas. Using pragmas for builds with the eMake Ninja emulation mode enabled (`--emake-emulation=ninja`) requires an addendum file that contains the pragmas for Ninja targets. Pragma addendum files are optional with any other emulation mode. For more information about pragma addendum files, see the "Specifying Pragmas in an Addendum File on page 4-13 section in the "Additional eMake Settings and Features" chapter.
`--emake-priority=<low|normal>`	–	Sets the priority for the current build's use of Agents in the cluster. When set to `normal`, the build uses at least the minimum number of Agents set in Cluster Manager. The default setting is determined by the Cluster Manager. This option overrides the priority associated with a build class but can be used to lower the build priority only.
`--emake-read-only=<path>`	EMAKE_READ_ONLY	All paths starting at the directories specified in `--emake-read-only` will be marked as read-only file systems when they are accessed on the agent. On UNIX, any attempt to create new files or write to existing files under those directories will fail with EROFS, "Read-only file system". On Windows, it will fail with ERROR_ACCESS_DENIED, "Access is denied."

Command-line Options	Environment Variables	Description
`--emake-readdir-conflicts=<0|1>`	–	Explicitly enables conflict detection on directory-read operations (commonly called "glob conflicts," which is only one manifestation of the problem). Allowed values are `0` (disabled, the default value) and `1` (enabled). If your build is susceptible to readdir conflict failures, you can enable these checks and get a correct build even if you do not conduct a single-agent build. The resulting history file is identical to a single-agent build result. Though the initial run with this feature might be over-serialized (a consequence of readdir conflicts), a good history file allows builds to run full speed, without conflicts, the next time. You do *not* want to enable this option all the time. Instead, you should enable it for one run if you suspect a globbing problem, and then disable it, but use the history file generated by the previous run. Another possible strategy to use, if you are not familiar with your build, is to enable the option until you get a successful build, and then disable it after you have a complete, good history file. **Note:** As an alternative, you can use the `#pragma readdirconflicts` pragma to enable directory-read conflicts on a per-job basis. You can apply it to targets or rules in your makefiles. It incurs less overhead than `--emake-readdir-conflicts=1` (which enables directory-read conflicts for an entire build). You can use this pragma in pragma addendum files as well as in standard makefiles.
`--emake-remake-limit=<N>`	–	This option defaults to `10`. If set to `0`, makefiles are not added as goals at all and no remaking occurs. Setting the value to `1` is equivalent to the deprecated `--emake-multiemake=0`.
`--emake-resource=<resource>`	`EMAKE_RESOURCE`	The resource requirement for this build. This option overrides the build class resource request setting set on the Cluster Manager.

Command-line Options	Environment Variables	Description

`--emake-root=<path>`	`EMAKE_ROOT`	Specifies the eMake root directory(s) location.
		
		Particularly on Windows, this parameter should ***not*** be used to virtualize your tool chain. Tools should be installed on each agent host for performance reasons and to avoid having to virtualize registry parts.
		
		The semi-colon is the delimiter between drives.
		
		Example:
		
		`build@winbuild-cm$ emake --emake-cm=winbuild-cm --emake-emulation=cygwin --emake-root=/c/cygwin/tmp;/c/tmp`
		
		`Starting build: 867`
		`make: Nothing to be done for `foo'.`
		`Finished build: 867 Duration: 0:00 (m:s)`
		`Cluster availability: 100%`
		
		In this example, the C: drive is mounted on /c.

`--emake-showinfo=<0|1>`	–	Turns build information reporting on (`1`=default) or off (`0`). Information includes build time to completion and average cluster utilization.
`--emake-sort-roots=<0|1>`	–	Enables (`1`) or disables (`0`) sorting of the eMake root directories in a history file, which are sorted alphabetically by default. This option lets you control whether the order of the directories in the history file follows the order specified via the `--emake-root` eMake option. The default is `1` (enabled). If you switch the value of the `--emake-sort-roots` option, you must first delete your history file and remove the entire contents of your eMake asset directory (the `.emake` directory by default). This deletes all data for JobCache, scheduling, and dependency optimization.

Command-line Options	Environment Variables	Description
`--emake-test-case-mode=` *<0|1>*	–	Turns test case mode on (`1`) or off (`0`). This option enables an execution mode specifically for test acceleration. The default is `0` (off). In test case mode, file system (and registry for Windows) updates from each job are discarded to disable conflict detection, history, and file-level annotation. This mode discards intermediate test results to restore agents to a pristine file system state after each test to ensure that each job runs as if no other tests ran. The results on `stdout` and `stderr` as well as the exit code are preserved. You must create a makefile specifically for this mode. You can use this mode with eMake in any emulation mode.
`--emake-tmpdir=` *<path>*	`EMAKE_TMPDIR`	Sets the eMake file temporary directory.
Caching Commands (Dependency Optimization, JobCache, Parse Avoidance, and Schedule Optimization)		
`--emake-assetdir=` *<path>*	–	Use the specified directory for cache locations for saved dependencies (for dependency optimization), JobCache, parse avoidance, and schedule optimization. The default name of this directory is .emake. By default, this directory is in the working directory in which eMake is invoked.
`--emake-jobcache=`*<cache_types>*	–	Enables the JobCache feature for all make invocations in a build. This option works for recursive and nonrecursive builds. You can specify any comma-separated list that includes one or more of the following values: `cl`, `clang`, `clang-cl`, `gcc`, `jack`, or `javac`. For details about these compiler options, see the " Job Caching on page 6-3" section.
`--emake-optimize-deps=` *<0|1>*	–	Use the saved dependency information file for a makefile when dependencies are the same and save new dependency information when appropriate.
`--emake-optimize-schedule=` *<0|1>*	–	Turns the schedule optimization feature on (`1`=default) or off (`0`). This feature uses performance and dependency information from previous builds to optimize the runtime order of jobs in subsequent builds.

Command-line Options	Environment Variables	Description
`--emake-parse-avoidance=<0|1>`	–	Avoid parsing makefiles when prior parse results remain up-to-date and cache new parse results when appropriate.
`--emake-parse-avoidance-ignore-env=<var>`	–	Ignore the named environment variable when searching for applicable cached parse results. To ignore more than one variable, use this option multiple times.
`--emake-parse-avoidance-ignore-path=<path>`	–	Ignore this file or directory when checking whether cached parse results are up-to-date. Append % for prefix matching. To ignore more than one path or prefix, use this option multiple times.
`--emake-suppress-include=<pattern>`	–	Skip matching makefile includes (such as generated dependencies). Generally, you should not suppress makefile includes unless they are generated dependency files, and you have enabled automatic dependencies as an alternative way of handling dependencies. **Note:** If the pattern does not have a directory separator, then the pattern is compared to the include's file name component only. If the pattern has a directory separator, then the pattern is taken relative to the same working directory that applies to the include directive and compared to the included file's entire path.
UNIX-Specific Commands		
`--emake-crossmake=<linux|solaris|solarisx86>`	–	Use this option to choose a specific operating system, so Cluster Manager will use Linux or Solaris hosts, specifically.

Command-line Options	Environment Variables	Description
–	`ECLOUD_ICONV_LOCALE`	Allows you to set the iconv locale. Use `iconv -l` to list the available locales. **Usage Note:** If you receive an emake assertion failure that contains information similar to: `emake: ../util/StringUtilities.h:406: std::string to_utf8(const std::string&): Assertion `c != iconv_t(-1)' failed.` This could mean that your system is missing an internationalization package, or the locale package has a different name for ISO 8859-1. (This issue is because of the conversion between 8-bit strings and UTF-8.) For example, your system recognizes “ISO8859-1”, “ISO_8859-1”, and “ISO-8859-1” only. Set `ECLOUD_ICONV_LOCALE` to one of those valid locale names.
Windows-Specific Commands		
`--emake-case-sensitive=<0|1>`	–	Sets case sensitivity for target and pattern name matching. This is inherited by all submakes in the build. The option applies when using gmake or cygwin emulation modes only; nmake and symbian modes are not affected. The default for gmake and cygwin is on (`1`).
`--emake-cygwin=<Y|N|A>`	`EMAKE_CYGWIN`	`Y`=requires *cygwin1.dll* `N`=ignore *cygwin1.dll* `A`=use *cygwin1.dll* if available Default=`A`, if launched from a Cygwin shell, if not, then `N`. Default=`Y`, if eMake `emulation=cygwin` was set.
`--emake-ignore-cygwin-mounts=<mounts>`	`EMAKE_IGNORE_CYGWIN_MOUNTS`	Comma-separated list of Cygwin mounts to ignore. Unless listed, Cygwin mount points are replicated on the Agent.
`--emake-reg-limit=<N>`	–	Limits the number of registry keys sent automatically for each key. Default=`50`.
`--emake-reg-roots=<path>`	–	Sets the registry virtualization path on Windows machines. The syntax is `--emake-reg-roots=path [;path]`. Do not use this parameter to virtualize your tool chain. Tools must be installed on each agent host for performance reasons and to avoid figuring out which registry parts to virtualize.

Command-line Options	Environment Variables	Description
Debug Commands		
`--emake-debug=<value>`	`EMAKE_DEBUG`	Sets the local debug log level(s). For a list of possible values, see the `emake --help` message.

`--emake-logfile=<file>`	`EMAKE_LOGFILE`	Sets the debug log file name.
		Default is `stderr`.
		The following macros are available:
		`@ECLOUD_BUILD_ID@` expands into the unique eMake build ID.
		`@ECLOUD_BUILD_TAG@` expands into the build tag from the Cluster Manager (this is displayed in the Name column on the Build tab in the Cluster Manager UI).
		`@ECLOUD_BUILD_DATE@` expands into an 8-digit code that represents the local system date where the build began, in the form YYYYMMDD.
		`@ECLOUD_BUILD_TIME@` expands into a 6-digit code that represents the 24-hour local system time where the build began, in the form HHMMSS.
		Example:
		`--emake-logfile=logfile-@ECLOUD_BUILD_ID@-@ECLOUD_BUILD_TAG@-@ECLOUD_BUILD_DATE@-@ECLOUD_BUILD_TIME@.xml`
		results in:
		`logfile-4-default_4_20090220184128-20090220-184128.xml`

`--emake-rdebug=<value>`	`EMAKE_RDEBUG`	Sets the remote debug log level(s). For a list of possible values, see the `emake --help` message. Enabling this option disables parse avoidance.

Command-line Options	Environment Variables	Description
`--emake-rlogdir=<dir>`	`EMAKE_RLOGDIR`	Sets the directory for remote debug logs.
Local Agent Commands		
`--emake-localagents=<y|n>`	–	Instructs eMake to use any available local agents . The default is: • `y` if you specified a Cluster Manager *or* if agents appear to be running • `n` if you did not request a Cluster Manager *and* it appears that local agents are unavailable If you have installed local agents (and they are currently running), but you do not want to use them, then specify `--emake-localagents=n` (with or without the `--emake-cm` option). This lets you use cluster agents only (without needing to shut down running local agents). If neither is specified but eMake detects that agents are running on the host, eMake uses them as if you specified `--emake-localagents=y`. The Cluster Manager prefers to allocate agents to eMake that are on the same host as eMake. You cannot use `--emake-cm` and `--emake-localagents=y` in the same eMake invocation. Local agents appear on the Cluster Manager with no special configuration needed. You can enable and disable local agents from cmtool just like remote agents. **Note:** "Sharing" local agents is only minimally supported without a Cluster Manager. You can run multiple eMake instances and explicitly limit them to a subset of the total local agents, but more dynamic sharing does not occur. If you want more sophisticated sharing, then you should use a Cluster Manager.

ElectricAccelerator Sample Build

After all ElectricAccelerator components are installed and you are familiar with the concepts, try a test build. Using a text editor, create a makefile with the following content:

UNIX

```
all: aa bb cc
aa:
	@echo building aa
	@sleep 10
bb:
	@echo building bb
	@sleep 10
cc:
	@echo building cc
	@sleep 10
```

Windows

```
SLEEP=ping -n 10 -w 1000 localhost>NUL
all: aa bb cc
aa:
	@echo building aa
	-$(SLEEP)
bb:
	@echo building bb
	-$(SLEEP)
cc:
	@echo building cc
	-$(SLEEP)
```

Note: “ping” is used in the Windows example because Windows does not have a SLEEP utility.

If you were to run this file with GNU Make, you would expect it to finish in approximately 30 seconds—allowing for each 10-second command to run serially. Running against an ElectricAccelerator cluster with at least three Agents, the commands run in parallel allowing the build to complete much faster.

To start this sample build:

- Specify the Cluster Manager by using the `--emake-cm=<host>` option. The Cluster Manager is responsible for assigning Agents to eMake for processing jobs. The example uses “linuxbuilder” as the Cluster Manager host.
- Make sure the eMake root directory [or directories] specification includes all directories that contain source or input files required by the build. In the example, the only source file is the makefile, which is in the same directory where eMake is invoked. Because the default emake root is the current directory, `--emake-root=<path>` is not needed.

```
% emake --emake-cm=linuxbuilder

Starting build: 1
building aa
building bb
building cc
Finished build: 1  Duration: 0:11(m.s) Cluster availability: 100%
```

Cluster availability: 100% indicates the cluster was fully available for the build duration. For more information on cluster sharing and the cluster availability metric, see Annotation on page 8-1.

Chapter 4: Additional eMake Settings and Features

The following topics discuss additional eMake settings and features.

Topics:

- Using Build Classes
- Using Priority Pools
- Using the Proxy Command
- Using Subbuilds
- Building Multiple Targets Simultaneously
- Using eMake Variables
- Using the Ninja Build System
- Specifying Pragmas in an Addendum File
- Terminating a Build
- Shutting Down Cluster Hosts During Builds

Using Build Classes

ElectricAccelerator provides features to organize builds in even the most complex environments. You do not need to create a build class—by default, a simple organizational structure is set up for you. But, if you have diverse product lines, or multiple product releases, you should set up and use build classes.

A *build class* is a flexible, user-defined classification for a designated group of builds. Using build classes is optional, but if you do not assign a build class, the Cluster Manager assigns the build to a default build class. Build classes help you organize the build management process.

Depending on your company requirements, you might use build classes to organize build groups by version or release, product type, development stage, or platform. You can decide how to use build classes to organize your builds into sets.

Classes have default priorities and boost values. Boost values have a range of -10 to +10 (default 0), where a higher value means that builds in that class can use available agents ahead of builds with the same priority but less boost.

Tag Definitions

Each build in a class is identified by a unique string called a *tag*. The build tag definition is a template that expands when a new build starts. The tag is user-defined and generally consists of a generic build name appended with build-specific data constructed from the following variables:

Tag	Description
`GC`	Globally unique number (Global Counter)
`LC`	Number unique to the build class (Local Counter; the build serial number within the class)
`BUILD_CLASS`	User-defined build class name
`BUILD_CLASS_ID`	System-generated number that the Cluster Manager uses to identify each class
`USER_NAME`	Name of the user who invoked eMake
`MACHINE_NAME`	Name of the machine where eMake was invoked
`USER_BUILD_LABEL`	Label specified at the eMake command line. For example, `--emake-build-label=my_build`
`BUILD_OS_ID`	Operating system ID under which the build was invoked. (`0` = undefined, `1` = Windows, `2` = Solaris, and `3` = Linux)

Tag	Description
DATE	Build start date and time using variables `Y, y, m d, H, M, S`. Ffor example, `2005-01-18 10:14:32` is `20050118101432`
.Y	Year at build start time (`YYYY`)
y	Year at build start time (`YY`)
m	Sequential month number at build start time (`1-12`)
d	Sequential day of month at build start time (`1-31`)
H	Hour of the day at build start time (`0-23`)
M	Minutes at build start time (`0-59`)
S	Seconds at build start time (`0-60`)
a	Abbreviated day of week at build start time (`WED`)
A	Full name of the day of week at build start time (such as `Wednesday`)
b	Abbreviated month name at build start time (such as `AUG`)
B	Full month name at build start time (such as `August`)
c	Build start date and time using the variables `A, B, d, H, M, S, Y`. For example, `2005-01-18 10:14:32` is `18/01/05 10:14:32`.

Together, the build name and variables are referred to as the *tag definition*. Variable names are case-sensitive.

For example, the tag definition `%BUILD_CLASS%_%LC%_%DATE%` for a build class named `QA_BUILD` creates the following build tag:

```
QA_BUILD_1234_20060123185958
```

When assigning build class tag definitions, choose from the list of tag variables above.

Build Class Examples

Suppose your company has two major product lines: SuperSoftware and MegaSoftware. SuperSoftware runs on Windows and Solaris platforms. MegaSoftware runs on Windows only. You could begin by setting up three classes that include the product name, the platform, and the current version number for each product:

- You could name the first class SuperSoftware_Win_v.2.1. The tag definition for this class would be:

  ```
  %BUILD_CLASS%_%LC%_%a%_%b%_%d%_%H%_%M%_%S%
  ```

The result would be a series of builds each named, or *tagged*, with the product name, the platform, the version number, a serial number (unique to the class), and the date for each build. For example:

```
SuperSoftware_Win_v.2.1_12345_WED_AUG_22_14_37_12
```

- The second class could be named SuperSoftware_Sol_v.1.7. The tag definition can be the same as in our first example, because it would be distinguished by the second build class name. Build tags in the second class would look like:

```
SuperSoftware_Sol_v.1.7_12356_WED_AUG_23_11_14_39
```

- The third class could be named MegaSoftware_Win_v.1.3. For this product, the tag definition would be similar to the previous examples but also could include the name of the user who started the build, because the MegaSoftware team is spread over several different locations. For this class, the tag definition might look like:

```
%BUILD_CLASS%_%LC%_%USER_NAME%_%DATE%
```

- As in the first two examples, the result would be a sequentially-numbered series of builds with the product name, platform, version number, name of the user who ran the build, and date of each build assigned through the build class:

```
MegaSoftware_Win_v.1.3_12356_JSMITH_20050411100838
```

Additional classes could be created when the development of SuperSoftware or MegaSoftware entered a new phase, such as a new platform release or a new version release. In this way, the builds for each stage of development can be segmented into logical sets for a more manageable and organized workflow.

Creating a New Build Class Using the Cluster Manager

1. Open a web browser and go to the Cluster Manager host.
2. Click the **Build > Build Classes** tab.

   The **Build Classes** page displays.
3. Click the **New Build Class** link.

   A blank class details screen appears.
4. Click the **Show Help** link on the right side of the screen to see field descriptions ,and then fill in the fields accordingly.
5. In the **Tag Definition** text box, enter the build class tag definition.

   To avoid errors, follow standard naming conventions for tag definitions by using numbers, letters, and underscores only without leading or trailing white spaces. Use underscores (_) instead of spaces. Use a percent sign on either side of any variables used. (For example, `%DATE%`).
6. Continue filling in the fields.

   See the online help for more information if needed.
7. Click **OK**.

Note: You can add comments to the build class. To do so, click the **Build > Build Class** tab, then select a build class name. For details about on adding or editing build class comments, adding or editing class properties, or deleting a build class, see the online help.

Using an Existing Build Class

Assign the build's class name through the `--emake-class=<`*`exact build class name`*`>` eMake command-line option when the build is invoked. If you do not assign a build class, the Cluster Manager assigns the build to the default class. If the class name typed on the eMake command line does not match a class name already in the Cluster Manager, eMake exits.

Makefile Macros

eMake automatically creates makefile macros (`ECLOUD_BUILD_CLASS` and `ECLOUD_BUILD_TAG`) from Cluster Manager build class data. These macros can be used to place generated values in your makefiles. For details, see the "Using eMake Variables on page 4-12" section.

Using Priority Pools

Priority Pools allow you to group resources into pools that can be prioritized differently among groups. Each pool's resources can potentially be used by any build, but builds originating from a pool's "owner" always have first priority to use that pool's resources. Using Priority Pools allows you to manage resource allocation for builds more efficiently.

Follow this procedure:

1. Enable Priority Pools through either the Cluster Manager or cmtool.
   - Using the Cluster Manager, go to **Administration > Server Settings** and select **Priority Pools**.
   - Using cmtool, run this command:

     ```
     cmtool modifyServer --resourceManagerType prioritypool
     ```

3. Define your pool using the **Agents > Resources** page. A pool resource uses this form: `__pool_xxx`, for example, `__pool_a`.

   Note: When you define a pool on the **Resources** page, you must include "__pool_". When you include a pool in the `--emake-resource` option, you can omit `__pool_` from the option.

4. Go to **Agent > Agent Policies** and set agent allocation policy to shared.
5. Launch eMake with the following option:

   ```
   --emake-resource="<pool resource>:<static resource>"
   ```

   Information about values for `--emake-resource=`:

   - When adding a pool resource to the option, you can omit `__pool_` from `__pool_xxx:` and use `xxx:` only.
   - A pool name (before the `:`) is not required. Not defining the pool name means the build will not use a pool resource.

- A resource name (after the :) is not required. Omitting resource name causes the build to attempt to use any unused resources that it is allowed to use.

When attempting to use a pool resource as a normal resource, there is a period (default is 60 seconds) during which, before your build starts, a build from the pool resource's owner can take back the resource.

Use Case 1: High Performance Builds

You have two pools of resources, one for high performance builds and one for low performance builds. You want to ensure that high performance builds can always use the more powerful 8-core machines and that low performance builds use the 4-core machines. You also want to allow high performance builds to use the 4-core machines when low performance builds are not running. And you want to allow low performance builds to use the 8-core machines when high performance builds are not running. You also have two special software packages, so you define a static resource for each.

Pool makeup:

- Pool a—High performance build resources, five 8-core machines, defined on the Resources page as *__pool_a*

  The machines are named h_1, h_2, h_3, s_1, and s_2.

- Pool b—Low performance build resources, five 4-core machines, defined on the Resources page as *__pool_b*

  The machines are named h_4, m_1, m_2, s_3, and s_4.

- Two static resources are also defined on the Resources page:
  - *s*—This resource includes these machines (which have a specific software package): s_1, s_2, s_3, and s_4 (from using host mask *s**)
  - *m*—This resource includes these machines (which have a specific software package): m_1 and m_2 (from using host mask *m_1, m_2*)

Launching eMake with --emake-resource=	**means that the build uses**
`"a:"`	Pool "a" and any unused resources
`"a:s"`	Static resource "s" (four machines). If a low performance build is running, only machines "s_1" and "s_2" (two machines) are used.
`"b:m"`	Static resource "m" only

`"a:m"`	Static resource “m” only. If a low performance build is running, no machines are available and the build must wait.

`":s"`
or
`"s"`
	Static resource “s” if low and high performance builds are not running. If a low performance build is running, then only machines “s_1” and “s_2” are used. If a high performance build is running, then only machines “s_3” and “s_4” are used. If low and high performance builds are running, no machines are available and the build must wait.

`":"`
or
`""`
	Any unused resources (this is considered a common build)

Use Case 2: Multiple Departments

There are two departments and each department has its own pool of machines. They want to contribute their machines toward a common pool so each department can use the other department’s machines while still ensuring that their own machines are available for their department’s builds. One department owns 20 machines and the other department owns 10 machines. Each department also owns a small number of 64-bit machines. IT contributes an additional four machines that any department can use.

Pool makeup:

- Pool Depta—Department A’s resources, 20 machines, defined on the Resources page as *__pool_Depta*

  The machines are named a_01 through a_16 and a_17_64 through a_20_64 (these last four machines are 64-bit).

- Pool Deptb—Department B’s resources, 10 machines, defined on the Resources page as *__pool_Deptb*

  The machines are named b_01 through b_08 and b_09_64 and b_10_64 (these last two hosts are 64-bit).

- Static resource *64bit*—This resource includes these machines: a_17_64 through a_20_64 and b_09_64 and b_10_64 (from using host mask *_64*).
- General build machines—IT department-supplied general build machines, four machines

Launching eMake with--emake-resource=	**means that the build uses**
`"Deptb:"`	Pool “Deptb” and any unused resources, using general build machines first and then Deptartment A’s machines if that department is not running a build.

`"Depta:"`	Pool "Depta" and any unused resources, using general build machines first and then Deptartment B's machines if that department is not running a build.
`"Depta:64bit"`	Static resource "64bit" (six machines). If Department B is running a build, only "a_17_64" through "a_20_64" (four machines) are used.
`":"` or `""`	Any unused resource, using general build machines first and then Deptartment A's and Deptartment B's machines if those departments are not running builds (this is considered a common build).

Using the Proxy Command

Normally, eMake sends commands to the agents for execution. In some cases, however, it might not be desirable or possible to execute a particular command on the agents.

Proxy Command Location

During installation, the `proxyCmd` binary is installed on every host:

- Linux: `<install_dir>/i686_Linux/bin/proxyCmd` where `<install_dir>` is `/opt/ecloud` by default
- Solaris: `<install_dir>/sun4u_SunOS/bin/proxyCmd` where `<install_dir>` is `/opt/ecloud` by default
- Windows: `<install_dir>\i686_win32\bin\proxyCmd.exe` where `<install_dir>` is `C:\ECloud` by default

When invoked by the Agent, it is: `proxyCmd <program> <arg1> ...`

ElectricAccelerator executes <*program*> on the host build system and proxies the result back to the Agent so it can continue remote execution.

Determine If You Can Use the Proxy Command

You can use the "proxy command" feature to run a command locally on the eMake machine *if both of the following are true*:

- You have a command that cannot run on the Agent, either because it returns incorrect results or because it is not available.
- The command in question *does not read or write build sources or output*—it only makes external [outside of the build] read-only queries.

The second item is particularly important because the command runs on the host build machine, outside of the virtualized file system. Because ElectricAccelerator cannot track the activity of this process, the dependency and conflict resolution mechanisms that prevent build output corruption are circumvented. It is also important that the process is read-only [that the command make no changes to whatever system it is querying] because in parallel builds with conflicts, eMake could rerun the job producing unintended side effects.

Note: For these reasons, it is important to use the "proxy command" only when necessary. Where possible, try to ensure cluster hosts have the same tools installed as the build system.

Proxy Command Example 1

The simplest *safe* use of the `proxyCmd` is a source control system query. For example, a particular build step queries the source control system for branch identification using a tool called `getbranch` that it embeds in a version string:

```
foo.o:
     gcc -c -DBRANCH=`getbranch` foo.c
```

It is preferable to avoid installing and configuring a full deployment of the source control system on the host when only this simple query command is needed.

In the following example, the `proxyCmd` provides an efficient solution. By replacing `getbranch` with `proxyCmd getbranch`, you avoid having to install the `getbranch` tool and its associated components on the host:

```
foo.o:
    gcc -c -DBRANCH=`proxyCmd getbranch` foo.c
```

Proxy Command Example 2

A less invasive implementation that does not require makefile modifications and allows compatibility with non-ElectricAccelerator builds is to create a link on all agent machines using the name of the tool [for example, 'getbranch'] found on the eMake host to `proxyCmd`. For Windows operating systems that do not support `symlinks`, use the `copy` command. When invoked under a different name, `proxyCmd` knows to treat the linked name as the <*program*> to execute:

```
# ln -s /opt/ecloud/i686_Linux/bin/proxyCmd /usr/bin/getbranch
```

Using Subbuilds

The eMake subbuild feature is designed to help speed up component builds through intelligent build avoidance. Currently, the subbuild feature's scope includes the following use case:

```
Makefile:
-------------
.PHONY: a b
all: a b
       @echo all
a b:
       $(MAKE) -C $@

a/Makefile
-------------
all: a.lib
       @echo a

a.lib:
       @touch $@

b/Makefile
-------------
```

```
all: ../a/a.lib
       @echo b
```

Explanation: If something from 'a' changes, and you are building from 'b', the only way to pick up the new `a.lib` is to build from the top level directory. With subbuilds, you know `b`'s dependencies so you can build those dependencies directly without having to build *everything* from the top level directory.

The subbuild database must be built beforehand to make the dependency list available without having to parse any Makefiles that are not in the current directory.

The following sections describe how to use subbuilds. Refer to Subbuild Limitations on page 4-10 for additional information about subbuild limitations.

Subbuild Database Generation

The following command runs your build as normal and also generates a subbuild database with the name `emake.subbuild.db`.

```
emake --emake-gen-subbuild-db=1 --emake-root=<path> --emake-subbuild-
db=emake.subbuild.db
```

Where <`path`> is the eMake root directory setting.

`--emake-root` is required for cluster builds.

`--emake-subbuild-db` is optional. If it is missing, the default `emake.subbuild.db` name is used.

Run a Build Using Subbuild

The following command runs a build with subbuild information:

```
emake --emake-subbuild-db=emake.subbuild.db
```

Specify `--emake-subbuild-db=<file>` to run a build with subbuild information. When you invoke eMake with the `--emake-subbuild-db` option, it uses the dependencies extracted from the makefile and the subbuild database to determine which build components are prerequisites of the desired current make, then rebuilds those components before proceeding as normal.

When you specify `--emake-subbuild-db=<file>`, do ***not*** specify `--emake-gen-subbuild-db`, otherwise eMake regenerates the database.

Subbuild Limitations

- There is no incremental building of the database. Each time you change something in a makefile in your build, you must rebuild the database by doing a full build.
- The database is not currently optimized for size. This might result in an extremely large database for very large builds.
- Subbuilds do not provide additional gains in non-recursive make builds.
- Because of the manner in which subbuilds are currently scheduled, there is interleaving output for the "Entering directory..." and "Leaving directory..." messages.

  For example: If a subbuild database was built for the following build:

```
Makefile:
-------------
.PHONY: a b
all: a b
a b:
       $(MAKE) -C $@

a/Makefile
----------------
all: a.lib
a.lib:
       echo aaa > $@

b/Makefile:
----------------
all: ../a/a.lib
       echo b
```

When you proceed to build just 'b' (maybe with "emake -C b") and a/a.lib is missing, you receive "entering directory a" after "entering directory b", even though 'a' is supposed to be built before 'b'.

```
make: Entering directory 'b'
make -C a
make[1]: Entering directory 'a'
echo aaa > a.lib
make[1]: Leaving directory 'a'
echo b
b
make: Leaving directory 'b'
```

Information Applying to Local Builds Only

- Rules to build a sub-directory's output files must not overlap.
  For example: The rule to build sub1/foo.o must appear in sub1/Makefile only and not sub2/Makefile. Default suffix rules can cause eMake to find a way to build sub1/foo.o while trying to build sub2. In this situation, adding ".SUFFIXES:" to sub2/Makefile can resolve the issue.
- Subbuilds require that the build be componentized to some degree.
- Subbuilds require that you have practiced "good hygiene" in your build tree—there must be explicit dependencies mentioned in the component makefiles.

  For example: If a build has two components, foo and bar, where foo produces a library foo.dll and bar uses that library, the rule might be written to produce bar.exe such as this in bar/Makefile:

```
bar.exe: $(BAR_OBJS)
       link $(BAR_OBJS) -l $(OUTDIR)/foo/foo.dll
```

  For subbuilds to work (in local mode), it must be modified as in the following:

```
bar.exe: $(BAR_OBJS) $(OUTDIR)/foo/foo.dll
       link $(BAR_OBJS) -l $(OUTDIR)/foo/foo.dll
```

  Note that it is explicitly stated that bar.exe requires foo.dll. Also note that it is NOT required to have a rule to build foo.dll in bar/Makefile.

There cannot be ANY rule at all to build `$(OUTDIR)/foo/foo.dll` in `bar/Makefile`, explicit or implicit, otherwise you will get the wrong information for building `foo/foo.dll` in the subbuilds database. The subbuilds database currently allows updates to existing entries while building the database.

Building Multiple Targets Simultaneously

The `#pragma multi` directive lets you use one rule that creates multiple outputs simultaneously. You can use this directive when a pattern rule might not be suitable. This directive lets you create non-pattern rules that have multiple outputs. You can use this directive only with gmake and NMAKE emulation.

`#pragma multi` causes the targets of the immediately-following rule or dependency specification to be treated as updated together if they are updated. For example, the following produces one rule and one rule job, rather than three of each:

```
#pragma multi
a b c: ; @echo building a b and c
```

The `#pragma multi` directive has the following restrictions:

- If you apply `#pragma multi` to a target list, then you must apply it to all overlapping target lists. Those lists must specify the same set of targets (although they might do so in a different order).
- Target- and pattern-specific variable assignments for the targets of a `#pragma multi` rule must agree. Otherwise, eMake might choose the assignments for just one target or combine them all.
- You cannot apply `#pragma multi` to static patterns, double-colon rules, or pattern targets that follow non-pattern targets on the same line.
- If you apply `#pragma multi` to a non-static pattern, a warning appears.
- A `#pragma multi` rule with commands might not override (or be overridden by) other commands for the same targets.
- A `#pragma multi` dependency specification must correspond to a `#pragma multi` rule with commands having the same set of targets. Otherwise, eMake will fail with errors. Implicit rules are not searched for these missing commands.
- `$@` has the same meaning as it does in multiple-target patterns: the target that first caused the rule to be needed.
- Setting the `--emake-disable-pragma=multi` or `--emake-disable-pragma=all` options disables `#pragma multi`.

Using eMake Variables

eMake automatically defines several variables that can be used in makefiles to access Cluster Manager-specific values during a build. For example, you could insert the build tag into your compilation step by typing:

```
main.o:main.cpp
        gcc -DBUILD_TAG="$(ECLOUD_BUILD_TAG)" -o main.o main.cpp
```

The variables eMake automatically creates are described in the following table.

Variable	Definition
`ECLOUD_BUILD_CLASS`	The build class as specified on the command line with `--emake-class`. For more information, see Using Build Classes on page 4-2.
`ECLOUD_BUILD_ID`	A globally unique build ID. This value is guaranteed unique across all builds in all classes for this Cluster Manager.
`ECLOUD_BUILD_TAG`	The tag as configured in Cluster Manager for this build class. For more information, see Using Build Classes on page 4-2.
`ECLOUD_BUILD_COUNTER`	A unique ID for this build within the current build class. This value is guaranteed to be unique across all builds in the same class, but not across builds in different classes.
`ECLOUD_BUILD_TYPE`	Allows you to discover the build type: *local* for a local build; or *remote* for a cluster build.

Using the Ninja Build System

To allow Android version 8 ("O") and Chromium builds, eMake supports the Ninja build system. eMake can emulate Ninja in addition to the standard Accelerator make-based builds such as gmake and NMAKE. eMake supports Ninja version 1.7.2 and newer.

eMake with Ninja emulation supports schedule optimization, JobCache, parse avoidance, Ledger, eDepend (autodep), and pragmas. Dependency optimization is not supported. You can use Ninja and gmake-based builds in a single eMake invocation.

eMake with Ninja emulation supports pragmas. Using pragmas for builds with the eMake Ninja emulation mode enabled (`--emake-emulation=ninja`) requires an addendum file that contains the pragmas for Ninja targets. You specify the file by using the `--emake-pragmafile` option. Pragma addendum files are optional with any other emulation mode. For more information about pragma addendum files, see the "Specifying Pragmas in an Addendum File on page 4-13 section in the "Additional eMake Settings and Features" chapter.

Ninja emulation works only with builds using Agents. eMake does not support Ninja's "debug" or "tool" options.

To enable Ninja emulation, use the following option to set the emulation type:

```
    --emake-emulation=ninja
```

Specifying Pragmas in an Addendum File

A pragma addendum file is a simple text file that lets you easily add pragmas (using `#pragma`) to a build without modifying makefiles or other files, such as when you do not "own" those files. A pragma addendum file uses a subset of make syntax to declare pragmas on targets within the build. This syntax lets you use existing makefiles or other files unmodified while avoiding inconvenient makefile syntax such as tabs versus spaces, dollar-sign escapes, and eval.

The pragma addendum file format is similar to the make format. You must specify the pragmas and then the target to which they apply. All text content is treated as literal strings.

Using pragmas for builds with the eMake Ninja emulation mode enabled (`--emake-emulation=ninja`) requires a pragma addendum file. Pragma addendum files are optional with any other emulation mode.

Restrictions

- emake allows only comments, `#pragma` declarations, and target names to the left of colons. Commands and prereqs are not allowed.
- You cannot use make-instance-level pragmas such as `allserial` and `#pragma jobcache parse`.
- You cannot use variables. Instead, you must use path names that are absolute or relative to the current working directory (instead of using `$(...)` variable expansions).

Supported Pragmas

The supported pragmas are:

Pragma	Description
`jobcache`	Enables the JobCache feature, which can substantially reduce compilation time. `jobcache parse` is not supported. For details, see the " Job Caching on page 6-3 section in the "Performance Optimization" chapter.
`noautodep`	Disables the eDepend (autodep) feature. For details, see the "ElectricAccelerator eDepend on page 7-2 section in the "Dependency Management" chapter.
`readdirconflicts`	Enables directory-read conflicts on a per-job basis. You can apply it to targets or rules. This pragma incurs less overhead than the `--emake-readdir-conflicts=1` eMake option (which enables directory-read conflicts for an entire build). For details about directory-read conflicts, see the "Conflicts and Conflict Detection on page 7-13" section in the "Dependency Management" chapter.
`runlocal`	Runs jobs locally on the host build machine instead of on an Agent in the cluster. For details, see the "Running a Local Job on the Make Machine on page 6-25 section in the "Performance Optimization" chapter.
`runlocal -repopulate` *`<dir1>`* `[... -repopulate` *`<dirN>`*`]`	Tells eMake where to find files created by jobs instead of looking in the current working directory by default. For details, see the "Running a Local Job on the Make Machine on page 6-25 section in the "Performance Optimization" chapter.
`runlocal sticky`	Causes all jobs after a certain point in the build to run locally. For details, see the " Running a Local Job on the Make Machine on page 6-25 section in the "Performance Optimization" chapter.

Examples

Following is an example pragma addendum file:

```
# system.img targets are runlocal because they do lots of I/O.
#
#pragma runlocal
out/target/product/generic/system.img:

#pragma runlocal
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img:

#pragma jobcache javadoc
out/target/common/docs/api-stubs-timestamp:
```

Following is an example pragma addendum file for Android 7.0.0:

```
#pragma runlocal
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img:

#pragma runlocal
out/target/product/generic/system.img:

#pragma jobcache kati
out/build-aosp_arm.ninja:

#pragma noautodep ./out/build_number.txt
out/target/common/docs/api-stubs-timestamp:

#pragma noautodep ./out/build_number.txt
out/target/common/docs/system-api-stubs-timestamp:

#pragma noautodep ./out/build_number.txt
out/target/common/docs/test-api-stubs-timestamp:

#pragma noautodep ./out/build_number.txt
out/target/product/generic/obj/ETC/system_build_prop_intermediates/build.prop:

#pragma noautodep ./out/build_number.txt
out/target/common/docs/apache-http-stubs-gen-timestamp:

#pragma noautodep ./out/build_number.txt
out/host/common/obj/JAVA_LIBRARIES/cts-tradefed_
intermediates/com/android/compatibility/SuiteInfo.java:
```

This example sets the `runlocal` pragma on `system.img` This pragma lets you avoid the heavy I/O required to run it on the remote Agent. The `jobcache` pragma enables kati job caching for `build-aosp_arm.ninja`. The `noautodep` pragmas prevent unnecessary rebuilds just because the build number changed.

Specifying the File to Use

You use the `--emake-pragmafile` option to specify the pragma addendum file to use. Following is an example eMake command line that uses a pragma addendum file:

```
emake --emake-emulation=ninja --emake-pragmafile=my_ninja_settings.pragma -f
mybuild.ninja
```

Stopping a Build

You can stop an in-progress build by using one of these methods:

- Press **Ctrl-C** from the terminal where you invoked eMake
- Use ElectricAccelerator's web interface
- Use **cmtool**—see the *cmtool Reference Guide* at http://docs.electric-cloud.com/accelerator_doc/AcceleratorIndex.html.

Cluster Manager terminates builds that seem to be hung. If the Cluster Manager does not receive a request from eMake for 60 seconds, it considers the build to be hung and terminates it.

Using the Cluster Manager to Stop a Build

1. Open the web interface by typing the Cluster Manager host name in the location bar of your browser window.
2. Click **Builds**.
3. Click **Stop Build** in the Action column on the row exhibiting your build ID and Name.

Using cmtool to Stop a Build

Note: This is an advanced option, which assumes you are already familiar with using command-line tools.

1. Request a list of running builds. The syntax is:

   ```
   % cmtool --cm <clustermanager:port> getBuilds --filter <field name>=<value>
   ```

   For example, if linuxbuilder is the Cluster Manager host name, type:

   ```
   % cmtool --cm linuxbuilder getBuilds --filter "result !=-1"
   ```

   A list of running builds will display accompanied by a number of attributes for each build—for example: Build ID, machine name, build class, owner, build start time, and so on. The Build ID is used to identify a build for termination. Also you can obtain additional information and/or include comments about the build. For example, to sort builds by start time and request only the first ten builds display, enter:

   ```
   % cmtool --cm linuxbuilder getBuilds --order Id --filter=”ID<11”
   ```

   To get failed builds:

   ```
   % cmtool --cm linuxbuilder getBuilds --order "start_time desc"--filter
   "result !=0"
   ```

2. After you determine which build you need to terminate, use this syntax to end the build:

   ```
   % cmtool --cm <clustermanager> stopBuild <buildId>
   ```

   For example, to end build 4458, you would type:

   ```
   % cmtool --cm linuxbuilder stopBuild 4458
   ```

Shutting Down Cluster Hosts During Builds

You can remove agent machines from the cluster during a build without requiring downtime or disabling Agents in the Cluster Manager, with the following advisories:

- If possible, shut down Agents before rebooting agent machines. Not shutting down Agents before a reboot might cause commands to fail unexpectedly, which could manifest in spurious build failures.
- After rebooting, agent machines that automatically start Agents will connect to the Cluster Manager and will be assigned builds and so on.

Chapter 5: Make Compatibility

ElectricAccelerator is designed to be completely compatible with existing Make variants it emulates. There are, however, some differences in behavior that might require changes to makefiles or scripts included in the build.

Almost all GNU Make and NMAKE options are valid for use with ElectricAccelerator. However, ElectricAccelerator does not support some GNU Make and NMAKE options.

The following topics documents those differences and what actions to take to ensure your build is compatible with ElectricAccelerator.

Topics:

- Unsupported GNU Make Options and Features
- Unsupported NMAKE Options
- Commands that Read from the Console
- Transactional Command Output
- Stubbed Submake Output
- Hidden Targets
- Wildcard Sort Order
- Delayed Existence Checks
- Multiple Remakes (GNU Make only)
- NMAKE Inline File Locations (Windows only)
- How eMake Processes MAKEFLAGS

Unsupported GNU Make Options and Features

Unsupported GNU Make Options

Option	eMake response if specified
`-d (debug)`	Error message
`-j (run parallel)`	Ignored
`-l (load limit)`	Ignored
`-o (old file)`	Error message
`-p (print database)`	Error message
`-q (question)`	Error message
`-t (touch)`	Error message

GNU Make 3.81 Support

eMake does not support the following GNU Make 3.81 features:

- Though `$(eval)` is allowed in rule bodies, any variables created by the `$(eval)` exist only in the scope of the rule being processed, ***not*** at the global scope as in GNU Make. For example, the following is not supported:

```
all:
        $(eval foo: bar)
```

- Using `$*` for an archive member target

GNU Make 3.82 Support

GNU Make 3.82 emulation supports the following GNU Make 3.82 features:

- Multi-line variable definition types
- Behavior in which pattern rule searches prefer the most-specific matching pattern, rather than the first matching pattern. For example, for a target {{foo.o}} and patterns {{%.o}} and {{f%.o}}, in that order, gmake 3.81 uses the {{%.o}} pattern, because it is the first match, but gmake 3.82 uses the {{f%.o}} pattern, because it is a more specific match

Other GNU Make 3.82 features are not supported.

GNU Make 4.0 and 4.1 Support

GNU Make 4.0 and 4.1 emulation supports the GNU Make 4.0 shell assignment feature, where `VAR!=`*`<command>`* is equivalent to `VAR:=$(`*`<shell command>`*`)`. Other GNU Make 4.0 features are not supported.

GNU Make 4.2 Support

GNU Make 4.2 emulation supports the GNU Make 4.2 `$(.SHELLSTATUS)` variable. This variable is set to the exit status of the last `!=` or `$(shell ...)` function invoked in this instance of Make. The value is `0` if successful or > `0` if not successful. The value is unset if the `!=` or `$(shell ...)` function was not invoked. Other GNU Make 4.2 features are not supported.

Unsupported NMAKE Options

- `/C` (Suppresses default output, including nonfatal NMAKE errors or warnings, timestamps, and NMAKE copyright message. Suppresses warnings issued by `/K`)
- `/T` (Updates timestamps of command-line targets (or first makefile target) and executes preprocessing commands, but does not run the build)
- `/NOLOGO` (Suppresses the NMAKE copyright message)
- `/ERRORREPORT` (If nmake.exe fails at runtime, sends information to Microsoft about these internal errors)

Commands that Read from the Console

When GNU Make or NMAKE invokes a makefile target command, that process inherits the standard I/O streams of the Make process. It is then possible to invoke commands that expect input during a build, either from the terminal or passed into the Make standard input stream. For example, a makefile such as:

```
all:
        @cat
```

could be invoked like this:

```
% echo hello | gmake
hello
```

More commonly, a command in a makefile might prompt the user for some input, particularly if the command encounters an error or warning condition:

```
all:
        @rm -i destroy

% gmake
rm: remove regular file `destroy'? y
```

Neither of these constructs is generally recommended because systems that require runtime user input are tedious to invoke and extremely difficult to automate.

Because Accelerator runs commands on cluster hosts where processes do not inherit the I/O streams and console of the parent eMake, makefiles (such as the examples above) with commands expecting interactive input are not supported.

In the majority of cases, tools that prompt for console input contain options to disable interactive prompting and proceed automatically. For example, invoking "`rm`" without the "`i`" enables this behavior. For those that do not, explicitly feeding expected input (either from a file or directly by shell redirection or piping) will suffice. For example:

```
all:
echo y | rm -i destroy
```

Finally, tools such as Expect (see https://www.nist.gov/services-resources/software/expect) that automate an interactive session can be used for commands that insist on reading from a console.

Transactional Command Output

eMake simulates a serial GNU Make or NMAKE build. Though eMake runs many commands in parallel, the command output (including text written to standard streams and changes to the file system, such as creating or updating files) appears serially in the original order without overlapping. This feature is called *transactional output* (or "serial order execution") and is unique to Accelerator among parallel build systems. This feature ensures standard output streams and underlying file systems always reflect a consistent build execution state, regardless of how many jobs eMake is actually running concurrently on the cluster.

Transactional output is achieved by buffering the results of every command until the output from all preceding commands is written. Buffering means that while the output *contents* on the standard streams matches GNU Make or NMAKE exactly, the *timing* of its appearance might be a little unexpected. For example:

- **"Bursty" output** – One of the first things you notice when running a build with Accelerator is that it appears to proceed in bursts, with many jobs finishing in quick succession followed by pauses. This type of output is normal during a highly parallel build because many later jobs might have completed and output is ready to be written as soon as longer, earlier jobs complete. The system remains busy, continuously running jobs throughout the build duration, even if the output appears to have paused momentarily.
- **Output follows job completion** – GNU Make and NMAKE print commands they are executing before they are invoked. Because eMake is running many commands in parallel and buffering results to ensure transactional output, command-line text appears with the output from the command *after* the command has completed. For example, the last command printed on standard output is the job that just completed, not the one currently running.
- **Batch output** – As a way to provide feedback to the user during a long-running execution, some commands might write to standard output continuously during their execution. Typically, these commands might print a series of ellipses or hash marks to indicate progress or might write status messages to standard error as they run. More commonly, a job might have several long-running commands separated with echo statements to report on progress during build execution:

For example, consider a rule that uses `rsync` to deploy output:

```
install:
        @echo "Copying output into destination"
        rsync -vr $(OUTPUT) $(DESTINATION)
        @echo "done"
```

With GNU Make, users first see the `Copying output` echo, then the state information from `rsync` as it builds the file list, copies files, and finally, they see the `done` echo as the job completes.

With eMake, all output from this job step appears instantaneously in one burst when the job completes. By the time any output from `echo` or `rsync` is visible, the entire job has completed.

Stubbed Submake Output

Recursive Makes (also called *submakes* because they are invoked by a parent or *top-level* Make instance) are often used to partition a build into smaller modules. While submakes simplify the build system by allowing individual components to be built as autonomous units, they can introduce new problems because they fragment the dependency tree. Because the top-level Make does not coordinate with the submake—it is just another command—it is unable to track targets across Make instance boundaries. For a discussion of submake problems, see "Recursive Make Considered Harmful" by Peter Miller (http://aegis.sourceforge.net/auug97.pdf)

Submakes are particularly problematic for parallel builds because a build composed of separate Make instances is unable to control target serialization and concurrency between them. For example, consider a build divided into two phases: a `libs` submake that creates shared libraries followed by `apps` that builds executables that link against those libraries. A typical top-level makefile that controls this type of build might look like this:

```
all: makelibs makeapps
makelibs:
	$(MAKE) -C libs
makeapps:
	$(MAKE) -C apps
```

This type of makefile works fine for a serialized make, but running in parallel it can quickly become trouble:

- If `makelibs` and `makeapps` run concurrently (as the makefile "all" rule implies), link steps in the `apps` Make instance might fail if they prematurely attempt to read `libs` generated libraries. Worse, they might link against existing, out-of-date library copies, producing incorrect output without error. This is a failure to correctly serialize dependent targets.
- Alternatively, if `apps` is forced to wait until `libs` completes, even `apps` targets that do not depend on `libs` (for example, all the compilation steps, which are likely the bulk of the build) are serialized unnecessarily. This is a failure to maximize concurrency.

Also important to note: Submakes are often spawned indirectly from a script instead of by makefile commands,

```
makelibs:
	# 'do-libs' is a script that will invoke 'make'
	do-libs
```

which can make it difficult for a Make system to identify submake invocations, let alone attempt to ensure their correct, concurrent execution.

These problems are exacerbated with distributed (cluster) parallel builds because each make invocation is running on a remote Agent.

Correct, highly concurrent parallel builds require a single, global dependency tree. Short of re-architecting a build with submakes into a single Make instance, this is very difficult to achieve with existing Make tools.

An ideal solution to parallelizing submakes has the following properties:

- maximizes concurrency, even across make instances
- serializes jobs that depend on output from other jobs
- minimizes changes to the existing system (in particular, does not require eliminating submakes or prohibit their invocation from scripts)

Submake Stubs

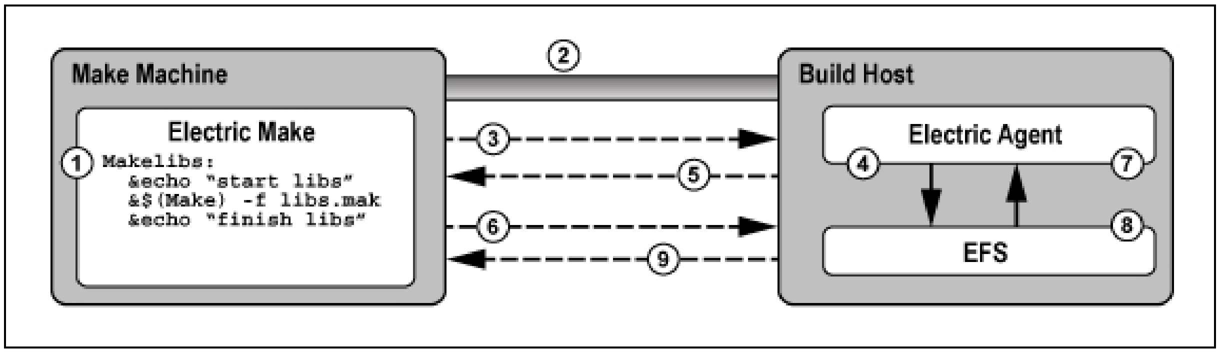
The parallel submake problem is solved by introducing submake stubs. eMake dispatches *all* commands, regardless of tool (compiler, packager, submake, script, and so on) to cluster Agents. After the Agent executes a command, it sends the results (output to standard streams and exit status) back to eMake, which then sends the next job command.

If the command run by the Agent invokes eMake (either directly by the expanded `$(MAKE)` variable or indirectly through a script that calls `emake`), a new top-level build is *not* started. Instead, an eMake process started on the Agent enters *stub* mode and it simply records details of its invocation (current working directory, command-line, environment, and so on) and immediately exits with status **0** (success) without writing output or reading any makefiles. The Agent then passes invocation details recorded by the stub back to the main eMake process on the host build machine, which starts a new Make instance and integrates its targets (which run in parallel on the cluster just like any other job) into the global dependency tree. Commands that follow a submake invocation are logically in a separate job serialized after the last job in the submake.

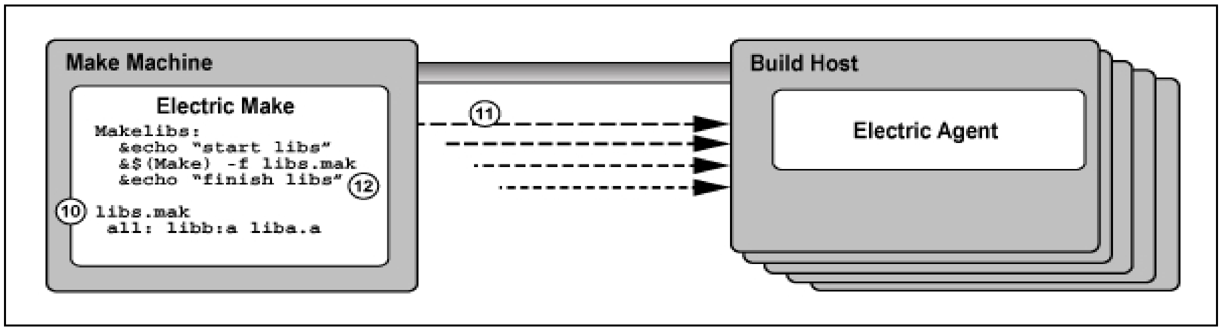
The following example illustrates this process:

```
...

Makelibs:
        @echo "start libs"
        @$(MAKE) -C libs
        @echo "finish libs"
```

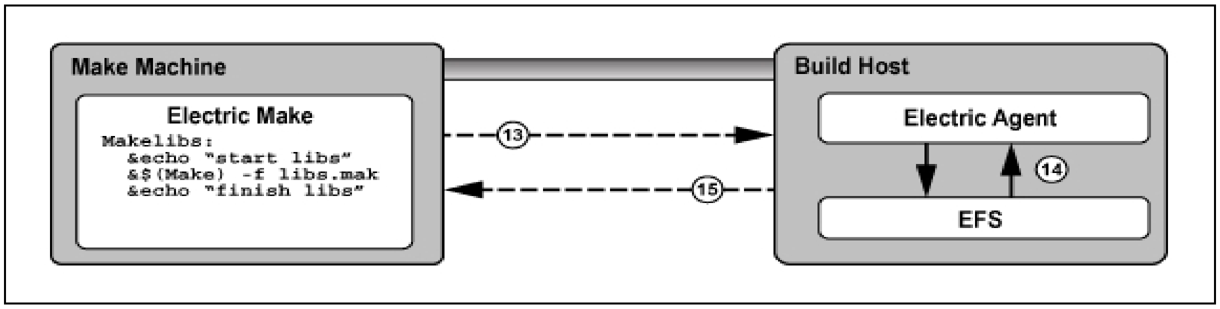
This example is diagrammed in steps as shown in the following illustrations.



1. eMake determines that `Makelibs` target needs to be built.
2. eMake connects to Electric Agent.
3. eMake sends the first command, `echo "start libs"`.
4. The Agent invokes the command, `echo "start libs"`, and captures the result.
5. The Agent returns the result, `"start libs"`, to eMake.
6. eMake sends the second command, `emake -f libs.mak`
7. The Agent runs the second command, `emake -f libs.mak`
8. `emake` enters stub mode and records the current working directory, command, and environment, then exits with no output and status code **0**
9. Agent returns the stub mode result and a special message stating a new Make was invoked with recorded properties (eMake).



10. eMake starts a new Make instance, reads the makefile, `libs.mak`, and integrates the file into the dependency tree.
11. New jobs are created and run against the cluster to build all targets in the `libs.mak` makefile.
12. eMake splits the last command in the `Makelibs` target, `echo "finish libs"`, into a continuation job that is defined to have a serial build order later than the last job in the submake, but there is no explicit dependency created that requires it to run later than any of the jobs in the submake. This means that it might run in parallel (or even before) the jobs in the submake, but if for some reason that is not safe, eMake will be able to detect a conflict and rerun the continuation job.



13. eMake sends the last command, `echo "finish libs"`.
14. Agent runs the command, `echo "finish libs"`, and captures the result.
15. Agent returns the result, `"finish libs"`, to eMake.

The eMake Stub solution addresses three basic parallel submake problems by:

- **Running parallel submakes in stub mode** – Stubs finish instantaneously and discover jobs as quickly as possible so the build is as concurrent as possible.
- **Creating a single logical Make instance** – New Make instances are started by the top-level eMake process only and their targets are integrated into the global dependency tree. eMake can then track dependencies across Make instances and use its conflict resolution technology to re-order jobs that might have run too soon.
- **Capturing submake invocations** – By capturing submake invocations as they occur, the submake stub works with your makefiles and builds scripts for the majority of cases "as-is." However, the stub introduces a behavior change that might require some makefile changes. See Submake Stub Compatibility on page 5-8.

Submake Stub Compatibility

Submake stubs allow your existing recursive builds to behave like a single logical Make instance to ensure fast, correct parallel builds. Stubs do introduce new behavior, however, that might appear as build failures. This section describes what constructs are not supported and what must be changed to make stubs compatible with submake stubs.

At this point, it is useful to revisit the relationship between commands and rules:

```
all:
       echo "this is a command"
       echo "another command that includes a copy" ; \
       cp aa bb
       echo "so does this command" ; \
       cp bb cc
       cp cc dd
```

The rule above contains four commands: (1) echo, (2) echo-and-copy, (3) another echo-and-copy, and (4) a copy. Note how semicolons, backslashes, and new lines delimit (or continue) commands. The rule becomes a job when Make schedules it because it is needed to build the "all" target.

Submake stubs' most important features:

- A submake stub never has any output and always exits successfully.
- The Agent sends stub output back (if any) after each command.
- Commands that follow a stub are invoked after the submake in the serial order.

Because a submake stub is really just a way of marking the Make invocation and does not actually do anything, *you cannot rely on its output (stdout/stderr, exit status, or file system changes) in the same command*.

In the following three examples, incompatible Accelerator commands are described and examples for fixing the incompatibility are provided.

Example 1: A command that reads submake file system changes

```
makelibs:
        $(MAKE) -C libs libcore.aa ; cp libs/libcore.aa /lib
```

In this example, a single command spawns a submake that builds `libcore.a` and then copies the new file into the `/lib` directory. If you run this rule as-is with Accelerator, the following error might appear:

```
cp: cannot stat 'libs/libcore.aa': No such file or directory
make: *** [all] Error 1
```

The submake stub exited immediately and the `cp` begins execution right after it in the same command. eMake was not notified of the new Make instance yet, so no jobs to build `libcore.a` even exist. The `cp` fails because the expected file is not present.

An easy fix for this example is to remove the semicolon and make the `cp` a separate command:

```
makelibs:
        $(MAKE) -C libs libcore.aa
        cp libs/libcore.aa /lib
```

Now the `cp` is in a command after the submake stub sends its results back to the build machine. eMake forces the `cp` to wait until the submake jobs have completed, thus allowing the copy to succeed because the file is present. Note that this change has no effect on other Make variants so it will not prevent building with existing tools.

Note: In general, Accelerator build failures that manifest themselves as apparent re-ordering or missing executions are usually because of commands reading the output of submake stubs. In most cases, the fix is simply to split the rule into multiple commands so the submake results are not read until after the submake completes.

Example 2: A command that reads submake *stdout*

```
makelibs:
         $(MAKE) -C libs mkinstall > installer.sh
```

The output is captured in a script that could be replayed later. Running this makefile with Accelerator always produces an empty `installer.sh` because submake stubs do not write output. When eMake does invoke this Make instance, the output goes to standard output, as though no redirection was specified.

Commands that read from a Make *stdout* are relatively unusual. Those that do often read from a Make that does very little actual execution either because it is invoked with `-n` or because it runs a target that writes to *stdout* only. In these cases, it is not necessary to use a submake stub. The Make instance being spawned is small and fast, and running it directly on the Agent in its entirety (instead of returning to the host build machine and distributing individual jobs back to the cluster) does not significantly impact performance.

You can specify that an individual `emake` invocation on the Agent does not enter stub mode, but instead behaves like a local (non-cluster) Make simply by setting the `EMAKE_BUILD_MODE` environment variable for that instance:

```
makelibs:
       EMAKE_BUILD_MODE=local \
$(MAKE) -C libs mkinstall >
       installer.sh
```

For Windows:

```
makelibs:
       set EMAKE_BUILD_MODE=local && $(MAKE) -C libs mkinstall >
               installer
```

eMake automatically uses local mode when the `-n` switch is specified.

Example 3: A command that reads submake exit status

```
makelibs:
        $(MAKE) -C libs || echo "failure building libs"
```

This example is a common idiom for reporting errors. The `||` tells the shell to evaluate the second half of the expression *only* if Make exits with non-zero status. Again, because a submake stub always exits with **0**, this clause will never be invoked with Accelerator, even if it would be invoked with GNU Make. If you need this type of fail-over handling, consider post-processing the output log in the event of a build failure. Also see Annotation on page 8-1 for more information.

Another common idiom in makefiles where exit status is read in loop constructs such as:

```
all:
       for i in dir1 dir2 dir3 ; do \
               $(MAKE) -C $$i || exit 1;\
       done
```

This is a single command: a "for" loop that spawns three submakes. The `|| exit 1` is present to prevent GNU Make from continuing to start the next submake if the current one fails. Without the `exit 1` clause, the command exit status is the exit status from the last submake, regardless of whether the preceding submakes succeeded or failed, or regardless of which error handling setting (for example, `-i`, `-k`) was used in the Make. The `|| exit 1` idiom is used to force the submakes to better approximate the behavior of other Make targets, which stops the build on failure.

On first inspection, this looks like an unsupported construct for submake stubs because exit status is read from a stub. Accelerator never evaluates the `|| exit 1` because the stub always exits with status code **0**. However, because the submakes really are reintegrated as targets in the top-level Make, a failure in one of them halts the build as intended. Explained another way, a submake loop is already treated as a series of independent targets, and the presence or absence of the GNU Make `|| exit 1` hint does not change this behavior. These constructs should be left as-is.

Hidden Targets

eMake differs from other Make variants in the way it searches for files needed by pattern rules (also called suffix or implicit rules) in a build.

- At the beginning of each Make instance, eMake searches for matching files for all pattern rules *before it runs any commands.* After eMake has rules for every target that needs updating, it schedules the rules (creating jobs) and then runs those jobs in parallel for maximum concurrency.
- Microsoft NMAKE and GNU Make match pattern rules *as they run commands*, interleaving execution and pattern search.

Because of the difference in the way eMake and NMAKE match pattern rules, NMAKE and eMake can produce different makefile output with *hidden targets*. A hidden target (also known as a "hidden dependency") is a file that is:

- created as an undeclared side-effect of updating another target
- required by a pattern to build a rule

Consider the following makefile example:

```
all: bar.lib foo.obj

bar.lib:
	touch bar.lib foo.c

.c.obj:
	touch $@
```

Notice that `foo.c` is created as a side effect of updating the `bar.lib` target. Until `bar.lib` is updated, no rule is available to update `foo.obj` because nothing matches the `.c.obj` suffix rule.

NMAKE accepts this construct because it checks for `foo.c` existence before it attempts to update `foo.obj`. NMAKE produces the following result for this makefile:

```
touch bar.lib foo.c
touch foo.obj
```

eMake, however, performs the search for files that match the suffix rule once so it can schedule all jobs immediately and maximize concurrency. eMake will not *notice* the existence of `foo.c` by the time it attempts to update `foo.obj`, even if `foo.c` was created. eMake fails with:

```
NMAKE : fatal error U1073: don't know how to make 'foo.obj'
Stop.
```

The fix is simply to identify `foo.c` as a product for updating the `bar.lib` target, so it is no longer a hidden target. For the example above, adding a line such as `foo.c: bar.lib` is sufficient for eMake to understand that `.c.obj` suffix rule matches the `foo.obj` target if `bar.lib` is built first. Adding this line is more accurate and has no effect on NMAKE.

GNU Make is similarly incompatible with eMake, but the incompatibility is sometimes masked by the GNU Make directory cache. GNU Make attempts to cache the directory contents on first access to improve performance. Unfortunately, because the time of first directory access can vary widely depending on which targets reference the directory and when they execute, GNU Make can appear to fail or succeed randomly in the presence of hidden targets.

For example, in this makefile, the file `$(DIR)/foo.yy` is a hidden target created as a side-effect of updating `aa` and needed by the pattern rule for `foo.xx`:

```
all: aa bb
aa:
        touch $(DIR)/foo.yy
bb: foo.xx

%.xx: $(DIR)/%.yy

        @echo $@
```

Depending on the value of `DIR`, this build might or might not work with GNU Make:

```
% mkdir sub; gmake DIR=sub
touch sub/foo.yy
foo.xx

% gmake DIR=.
touch ./foo.yy
gmake: *** No rule to make target 'foo.xx', needed by 'bb'.  Stop.
```

eMake does not attempt to emulate this behavior. Instead, it consistently refuses to schedule `foo.xx` because it depends on a hidden target (just as it did in the NMAKE emulation mode in the earlier example). In this case, adding a single line declaring the target: `$(DIR)/foo.yy: aa` is sufficient to ensure it always matches the `%.xx` pattern rule.

Note: If a build completes successfully with Microsoft NMAKE or GNU Make, but fails with "don't know how to make `<x>`" with eMake, look for rules that create `<x>` as a side-effect of updating another target. If `<x>` is required by a suffix rule also, it is a hidden target and needs to be declared as explicit output to be compatible with eMake.

There are many other reasons why hidden targets are problematic for all Make-based systems and why eliminating them is good practice in general. For more information, see:

- "Paul's Rules of Makefiles" by Paul Smith at http://www.make.paulandlesley.org/rules.html. Among other useful guidelines for writing makefiles, the primary author of GNU Make writes, "Every non-`.PHONY` rule **must** update a file with the *exact* name of its target. [...] That way you and GNU Make always agree."
- "The Trouble with Hidden Targets" by John Graham-Cumming at http://www.cmcrossroads.com/content/view/6519/120/.

Note: In a limited number of cases, eMake might conclude that a matching pattern rule for an output target does not exist. This occurs because eMake's strict string equality matching for prerequisites determines that the prerequisites are different (even though the paths refer to the same file) and that there is no rule to build it.

Wildcard Sort Order

A number of differences exist between GNU Make and eMake regarding the use of `$(wildcard)` and prerequisite wildcard sort order functions. When using the `$(wildcard)` function or using a wildcard in the rule prerequisite list, the resultant wildcard sort order might be different for GNU Make and eMake.

Different GNU Make versions are not consistent and exhibit permuted file lists. Even a GNU Make version using different system libraries versions will exhibit inconsistencies in the wildcard sort order.

No difference exists in the file list returned, other than the order. If the sort order is important, you might wrap `$(wildcard)` with `$(sort)`.

For example:

```
$(sort $(wildcard *.foo))
```

Do not rely on the order of rule prerequisites generated with a wildcard. For example, using `target: *.foo`.

Relying on the order of `*.foo` can be dangerous for both GNU Make and eMake. Neither GNU Make nor eMake guarantees the order in which those prerequisites are executed.

Delayed Existence Checks

All Make variants process makefiles by looking for rules to build targets in the dependency tree. If no target rule is present, Make looks for a file on disk with the same name as the target. If this *existence check* fails, Make notes it has no rule to build the target.

eMake also exhibits this behavior, but for performance reasons it delays the existence check for a target without a makefile rule until just before that target is needed. The effect of this optimization is that eMake might run more commands to update targets (than GNU Make or NMAKE) *before* it discovers it has no rule to make a target.

For example, consider the following makefile:

```
all: nonexistent aa bb
aa:
       @echo $@
bb:
       @echo $@
```

GNU Make begins by looking for a rule for `nonexistent` and, when it does not find the rule, it does a file existence check. When that fails, GNU Make terminates immediately with:

```
make: *** No rule to make target 'nonexistent', needed by 'all'.  Stop.
```

Similarly, NMAKE fails with:

```
NMAKE : fatal error U1073: don't know how to make 'nonexistent' Stop.
```

eMake delays the existence check for `nonexistent` until it is ready to run the `all` target. First, eMake finishes running commands to update the `aa` and `bb` prerequisites. eMake fails in the same way, but executes more targets first:

```
aa
bb
make: *** No rule to make target 'nonexistent', needed by 'all'.  Stop.
```

Of course, when the existence check succeeds (as it does in any successful build), there is no behavioral difference between eMake and GNU Make or Microsoft NMAKE.

Multiple Remakes (GNU Make only)

GNU Make has an advanced feature called Makefile Remaking, which is documented in the GNU Manual, "How Makefiles are Remade," and available at:
http://www.gnu.org/software/make/manual/make.html#Remaking-Makefiles

To quote from the GNU Make description:

> "Sometimes makefiles can be remade from other files, such as RCS or SCCS files. If a makefile can be remade from other files, you probably want make to get an up-to-date version of the makefile to read in.
>
> "To this end, after reading in all makefiles, make will consider each as a goal target and attempt to update it. If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit rule applies to it (see section Using Implicit Rules), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, make starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.)"

This feature can be very useful for writing makefiles that automatically generate and read dependency information with each build. However, this feature can cause GNU Make to loop infinitely if the rule to generate a makefile is always out-of-date:

```
all:
       @echo $@
makefile: force
       @echo "# last updated: 'date'" >> $@
force:
```

In practice, a well-written makefile will not have out-of-date rules that cause it to regenerate. The same problem, however, can occur when Make detects a clock skew—most commonly due to clock drift between the system running Make and the file server hosting the current directory. In this case, Make continues to loop until the rule to rebuild the makefile is no longer out-of-date.

In the example below, `DIR1` and `DIR2` are both part of the source tree:

```
-include $(DIR1)/foo.dd
all:
       @echo $@
$(DIR1)/foo.dd: $(DIR2)/bar.dd
%.d:
       touch $@
```

If two directories are served by different file servers and the clock on the system hosting `DIR2` is slightly faster than `DIR1`, then even though `foo.dd` is updated after `bar.dd`, it might appear to be older. On remaking, GNU Make will again see `foo.dd` as out-of-date and restart, continuing until the drift is unnoticeable.

This problem is particularly troublesome on Solaris, where GNU Make timestamp checking has nanosecond resolution:

```
% make
touch dir2/bar.dd
touch dir1/foo.dd
gmake: *** Warning: File 'bar.dd' has modification time in the future (2005-04-
11 13:52:46.811724 > 2005-04-11 13:52:46.799573198)
touch dir1/foo.dd
touch dir1/foo.dd
all
```

eMake fully supports makefile remaking and can be configured to behave exactly as GNU Make. However, by default, to ensure builds do not loop unnecessarily while remaking, eMake limits the number of times it restarts a make instance to 10. If your build is looping unnecessarily, you might want to lower this value or disable remaking entirely by setting:

```
--emake-remake-limit=0
```

NMAKE Inline File Locations (Windows only)

NMAKE contains a feature to create *inline files* with temporary file names. For example, the following makefile creates a temporary inline file containing the word “pass” and then uses “type” to output it.

```
all:
               type <<
       pass
       <<
```

With Microsoft NMAKE, the file is created in the directory where `%TMP%` points. eMake does not respect the `%TMP%` setting and creates the inline file in the rule working directory that needs the file.

How eMake Processes MAKEFLAGS

eMake uses the following process:

1. Similar to GNU Make, eMake condenses no-value options into one block.
2. When eMake encounters an option with a value, it does what GNU Make does, it appends the value and starts the next option with its own -
3. Certain options are ignored/not created. This changes the layout of the options in `MAKEFLAGS` (for example `-j`, `-l`).
4. eMake-specific options are not added to `MAKEFLAGS`, but are handled through `EMAKEFLAGS`.
5. Passing down environment variables as `TEST=test` renders the same result as in GNU Make (an extra - at the end, followed by the `variable=value`).
6. On Windows, eMake prepends the `--unix` or `--win32` flag explicitly.

Chapter 6: Performance Optimization

The following topics discuss how to use ElectricAccelerator performance optimization features and performance tuning techniques.

Topics:

- Optimizing Android Build Performance on page 6-2
- Dependency Optimization on page 6-2
- Job Caching on page 6-3
- Parse Avoidance on page 6-17
- Javadoc Caching on page 6-23
- Schedule Optimization on page 6-24
- Running a Local Job on the Make Machine on page 6-25
- Serializing All Make Instance Jobs on page 6-26
- Managing Temporary Files on page 6-28

Optimizing Android Build Performance

The information in this section is replaced by the *KBEA-00165 - Best Practices for Android Builds Using ElectricAccelerator 10.1* KB article.

Dependency Optimization

ElectricAccelerator includes a dependency optimization feature to improve performance. By learning which dependencies are actually needed for a build, Accelerator can use that information to improve performance in subsequent builds.

When dependency optimization is enabled, eMake maintains a dependency information file for each makefile. If a build's dependencies have not changed from its previous build, eMake can use that stored dependency information file for subsequent builds.

Enabling Dependency Optimization

You must first run a "learning" build with the dependency optimization feature enabled. To enable dependency optimization, set the following: `--emake-optimize-deps=1`

For the learning build, (because there is no stored dependency information file for that specific makefile) the argument only saves a new result. For subsequent builds, the argument enables the reuse of stored dependency information and saves a new file as appropriate. If you do not specify `--emake-optimize-deps=1`, then dependency information is not saved or accessed.

The following table describes dependency optimization-related options.

eMake Option	Description
`--emake-assetdir=<`*`path`*`>`	Use the specified directory for assets such as saved dependency information. The default directory is named .emake. (This option also determines the cache locations for the parse avoidance feature and for JobCache.)
`--emake-optimize-deps=<`*`0/1`*`>`	Use the saved dependency information file for a makefile when dependencies are the same and save new dependency information when appropriate.

Dependency Optimization File Location and Naming

Dependency information files are saved in the working directory where eMake was invoked under `<`*`assetdir`*`>/deps`. (The default asset directory is `.emake`.)

The file is named from the MD5 hash of the root-relative path of the "main" makefile for the make instance that the dependency information file belongs to.

Examples

If your eMake root is `/tmp/src`, and your makefile is `/tmp/src/Makefile`, then the dependency information file is named after the MD5 hash of the string "Makefile".

If your root is `/tmp/src` and your makefile is `/tmp/src/foo/Makefile`, the file is named after the MD5 hash of the string "foo/Makefile".

Job Caching

JobCache is a feature that can substantially reduce compilation time. JobCache lets a build avoid recompiling object files that it previously built, if their inputs have not changed. JobCache works even after you clean the build output tree (for example, by using "make clean"). By caching and reusing object files, JobCache can significantly speed up full builds.

JobCache uses cache "slots." When JobCache is enabled, eMake maintains a slot for each combination of command line options, relevant environment variable assignments, and current working directory. A slot can be empty or can hold a previously-cached result. If the appropriate slot holds an up-to-date result, a cache "hit" occurs, and compilation is avoided.

A cached result becomes obsolete if eMake detects file system changes that might have caused a different result (with the same command line options, environment variable assignments, and current working directory). Such file system changes include any files that are read during compilation, which means all source files, gcc precompiled headers, and compilation tools included in the eMake root.

Electric Cloud recommends that all components (Cluster Manager, Electric Agent/EFS, and eMake) on all machines in the cluster are upgraded to the latest version. However, you can still use JobCache with Agents on Electric Agent/EFS machines running versions as old as 7.2, as long as an appropriate backward compatibility package (BCP) is installed on those Electric Agent/EFS machines. For more information, see the *ElectricAccelerator Installation and Configuration Guide* at http://docs.electric-cloud.com/accelerator_doc/AcceleratorIndex.html.

Topics:

- Benefits
- Limitations
- Supported Tools
- Configuring JobCache
- Running a "Learning" Build to Populate the Cache
- Extending JobCache to Teams Via a Shared Cache
- Job Caching for gcc, clang, Jack, and javac
- Job Caching for cl
- Viewing JobCache Metrics
- Moving Your Workspace
- Deleting the Cache

Benefits

- Speeds long, full builds (for example, when you do a "make clean" then a "make," or when you run a build in a new workspace)
- Builds faster than ccache
- Avoids certain false cache hits that might occur when you use ccache

Limitations

- JobCache does not cache results from compilation jobs that invoke eMake. JobCache stores results from particular compilation jobs, but if a rule includes an invocation of eMake, (which might spawn other jobs), then that job is not cached, as in the following example:

  ```
  gcc -o foo.o foo.c && $(MAKE) -C subdir
  ```

- The date and time in an object file will still reflect the original compilation (not the current date or time) if you use a C preprocessor macro that expands to the date or time when the compiler runs, and ElectricAccelerator re-uses the resulting object file in a subsequent build.
- Source paths embedded in debugging information in object files will reflect the original compilation (even if you re-use the object file while building in a different workspace).

Supported Tools

JobCache supports the following tools.

Tool	Supported Platforms	Notes
gcc and g++	Linux Windows	• All .o and .lo files are cached • Should not be used to cache results from linking
clang and clang++	Linux Windows	• All .o and .lo files are cached • Should not be used to cache results from linking
cl (Microsoft Visual C/C++)	Windows	• All .obj files are cached • The only supported debug option is /Z7 • Should not be used to cache results from linking
Java Android Compiler Kit (Jack)	Linux	• All .jack and .dex files are cached
javac	Linux	• All .jar files are cached

Running a "Learning" Build to Populate the Cache

You must first populate the cache by running a "learning" build with JobCache enabled. For the learning build (because the cache is empty), JobCache saves only a new result to the cache. For subsequent builds, JobCache re-uses cached results and saves a new result to the cache as appropriate. If you do not enable JobCache, then the job cache is not accessed.

Extending JobCache to Teams Via a Shared Cache

The Shared JobCache feature extends JobCache to teams of developers by using a shared cache. As with non-shared JobCache, Shared JobCache accelerates builds by reusing outputs from a build in the next build, which avoids costly redundant work across builds. Shared JobCache extends this concept by giving developers read-only access to a cache that was previously populated by another user or process (such as a nightly build). With this feature enabled, only one user must actually run the compilations; other team members simply reuse the output from that “golden” build.

A shared cache gives JobCache two tiers: shared and private (“local”). The shared cache strictly augments the traditional (“local”) cache but does not replace it.

Shared JobCache tries to find matching cache entries in the following order:

1. During build execution with Shared JobCache enabled, eMake tries to find matching cache entries in the shared asset directory. This directory is specified by the `--emake-shared-assetdir` option. For example, `--emake-shared-assetdir=/net/nightlybuild/`.

   Developers can never modify cache entries in the shared asset directory.
2. If there is no matching slot for a job in the shared asset directory, or if the input files for the job differ, eMake uses the developer's local cache instead by checking for a hit there. This cache is specified by the `--emake-assetdir=<directory>` option. For example, `--emake-assetdir=/net/home/bob/`.
3. If there is no local cache match, eMake creates a slot in the developer's local cache if needed. The developer will get a hit in their local cache during the next compilation.

Prerequisites

Shared JobCache requires all participating developers to have access to a shared file system (such as NFS) where the shared asset directory will reside.

Populating the Shared Cache

For the builds that populate the shared cache, use the following eMake options:

```
--emake-jobcache=<types> --emake-assetdir=<assetdir>
```

Using the Shared Cache

Developer builds use the following eMake options:

```
--emake-jobcache=<types> --emake-shared-assetdir=<assetdir>
```

where *`<assetdir>`* is the asset directory of the populated cache.

Configuring JobCache

Licensing

JobCache is licensed based on the maximum number of builds that may use it simultaneously. This number is read from the `jobcacheMax` property in the Accelerator license file. Simultaneous builds that exceed this number occur without using this feature.

If the JobCache license entry is invalid, or if the number of simultaneous builds has exceeded the license limit, a `WARNING EC1181: Your license does not permit object caching` message appears when a build tries to use JobCache. eMake will continue to work normally.

Choosing a Disk for the Job Cache

To estimate the disk space required for the job cache, add the sizes of your object files together and multiply by 0.7. A specific example is that the Android KitKat Open Source Project (AOSP) requires about 4 GB. For best performance, choose a disk that is local to the eMake client host. For Shared JobCache, users must have access to a shared file system such as NFS.

You can use the `--emake-assetdir=` eMake option to specify the directory for your job cache. The default name of this directory is .emake. By default, this directory is in the working directory in which eMake is invoked. (This option also determines the cache location for the parse avoidance feature and the location of the saved dependency information for the dependency optimization feature.)

Building Multiple Branches

To maximize your cache hits, use the `--emake-assetdir=` option to specify a separate asset directory for each branch of code that you build.

Setting the eMake Root

JobCache does not detect changes to compilation inputs that are not under your eMake root. You must ensure that your eMake root contains all sources and tools that might change.

Job Caching for gcc, clang, Jack, and javac

Enabling JobCache for All Make Invocations in a Build

To enable JobCache for all make invocations in a build, use the `--emake-jobcache=<`*`types`*`>` eMake option, where `<`*`types`*`>` is a comma-separated list of any combination of `gcc`, `clang`, `clang-cl`, `jack`, or `javac`. The list cannot contain spaces. The `--emake-jobcache=<`*`type`*`>` option works for recursive and nonrecursive builds.

Notes:

- `--emake-jobcache=clang` is an alias for `--emake-jobcache=gcc`. In the eMake annotation file, the JobCache type appears as `jobcache type="gcc"`.
- `--emake-jobcache=clang-cl` is an alias for `--emake-jobcache=cl`. In the eMake annotation file, the JobCache type appears as `jobcache type="cl"`.

Following is an example command that enables JobCache for gcc, Jack, and javac:

```
% emake --emake-cm=mycm --emake-root=/src/mysource --emake-jobcache=gcc,jack,javac
--emake-annodetail=basic
```

If some of your makefile targets are built by rules that do not invoke gcc/g++, clang/clang++, Jack, or javac, and those rules do not behave enough like those tools for job caching to be suitable, then use the `jobcache` pragma to be more selective about which jobs are cached, either by enabling JobCache for fewer jobs or by selectively disabling it for particular jobs with `#pragma jobcache none`.

For example, the rules that build a particular ".o" file might use an environment variable that is not used by gcc/g++, and you might want to miss the cache if that environment variable changes its value.

The `--emake-annodetail=basic` option is not required to invoke eMake, but it is recommended for troubleshooting. For details, see the Troubleshooting section below.

For information about how to use this functionality with Visual Studio, see the Job Caching for cl When Using Visual Studio section.

Using the Response File gcc Command Line Option

JobCache supports the response file (`@<file>`) gcc command line option, which reads gcc command-line options from a separate file specified by <*file*>.

Enabling JobCache for All Object File Targets in a Make Invocation

If you do not want to enable JobCache for all make invocations in a build, you can still enable JobCache for specific object file targets by applying the `jobcache` pragma inside the makefile. To enable the feature for all targets with a particular extension, use the following:

```
#pragma jobcache <type> -exist *
%.<object_file_extension> :
```

where `<type>` is `gcc`, `clang`, `clang-cl`, `jack`, or `javac`.

Following is an example makefile excerpt with the feature enabled for all targets ending in ".o":

```
...
#pragma jobcache gcc -exist *

%.o :
...
```

(You can apply the pragma to a pattern, but not to a suffix rule.) If cache misses occur because an insignificant file exists or does not exist, you can delete `-exist *`, and eMake will check only the existence of files whose names suggest that they are source files or gcc precompiled headers. You can augment the set of files that matters to eMake by adding `-exist` options that specify the desired glob patterns.

If you do not want to add the pragma to the main makefile, you can add it to an "addendum" makefile. This is a small makefile that you include in the eMake invocation from the command line. For example, you could create a makefile named `jobcache.mak` and then add `-f Makefile -f jobcache.mak` to your eMake invocation. An addendum makefile is useful when you do not control the content of your makefiles (such as when you include open source components that use build tools such as configure, CMake, or qmake to generate makefiles).

Enabling or Disabling JobCache for Specific Object File Targets

To enable JobCache only for some object file targets, do one of the following:

- If an object file is built by an explicit rule (one without "%" in it), you can enable JobCache for that rule by inserting a pragma just above it in the makefile. You can also use an addendum makefile in this case. For example, when using gcc/g++, clang/clang++, Jack, or javac to cache an explicit rule producing target.o, you can use this addendum:

  ```
  #pragma jobcache type -exist *
  target.object_file_extension :
  ```

  For example, to enable JobCache using gcc only for some object file targets:

  ```
  #pragma jobcache gcc -exist *
  target.o :
  ```

- You can use more narrow target patterns than "%.o" after a `jobcache` pragma. For example, "abc%.o" applies the pragma to "abcdef.o" but not to "xyzdef.o". (This is true even if the rule to build "abcdef.o" appears elsewhere in the makefile.)

Similarly, you can explicitly disable caching by using the `#pragma jobcache none` pragma.

If `jobcache` pragmas with different options apply to a target, then eMake selects one of them as follows:

- If the target is built by an explicit rule with a `jobcache` pragma, then eMake chooses that pragma.
- Otherwise, eMake chooses the pragma whose pattern has the most in common with the target name.
- If eMake still encounters ties, then it chooses the pragma that it encountered last during makefile parsing.

For example, the following gcc makefile applies job caching to hello1.o and hello2.o, but not to hello3.o or date2.o:

```
CC=gcc

all : hello
       ./hello

#pragma jobcache gcc -exist *
hello1.o: hello1.c hello1.h
       $(CC) -c -o $@ $< $(CFLAGS)

hello2.o: hello2.c hello2.h
       $(CC) -c -o $@ $< $(CFLAGS)
# Uses __DATE__ to record when build occurred--do not cache.
#pragma jobcache none
date2.o: date2.c date2.h
       $(CC) -c -o $@ $< $(CFLAGS)

hello3.o: hello3.c hello3.h
       $(CC) -c -o $@ $< $(CFLAGS)
```

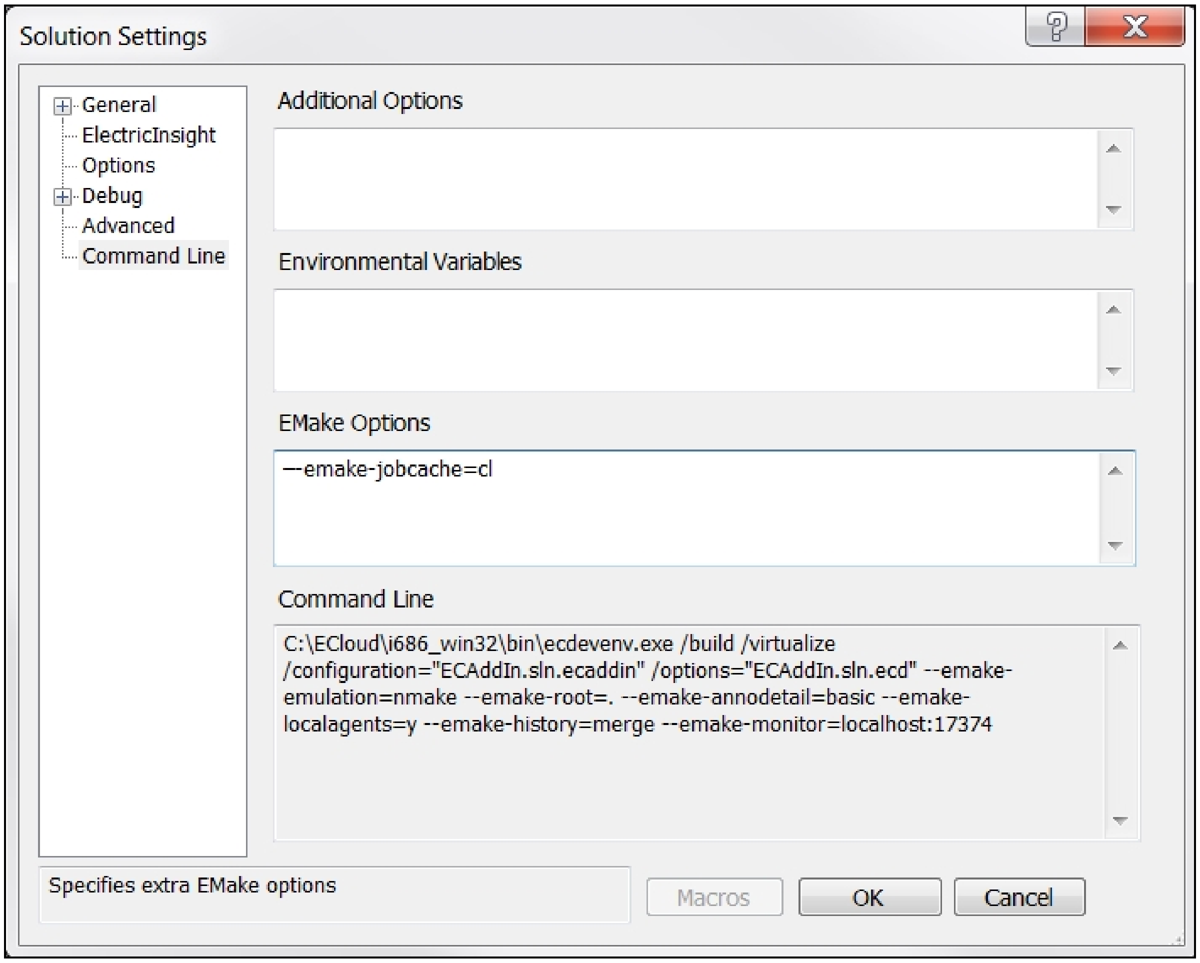
```
hello: hello1.o hello2.o hello3.o
       $(CC) -o $@ $^

clean:
       rm hello hello1.o hello2.o hello3.o

.PHONY: all clean

#pragma jobcache gcc -exist *
%2.o :
```

Job Caching for cl

Enabling cl JobCache for All Make Invocations in a Build

On Windows, setting `–emake-jobcache=cl` enables JobCache for all jobs that produce `.obj` files. These are normally created by the Visual C++ compiler (`cl.exe`). You can set the option on the command line, in an emake.conf file, or in the `EMAKEFLAGS` environment variable.

When using the Visual Studio IDE extension, add the `--emake-jobcache=cl` option to EMake Options in the command line settings:



Additional Options

Environmental Variables

EMake Options

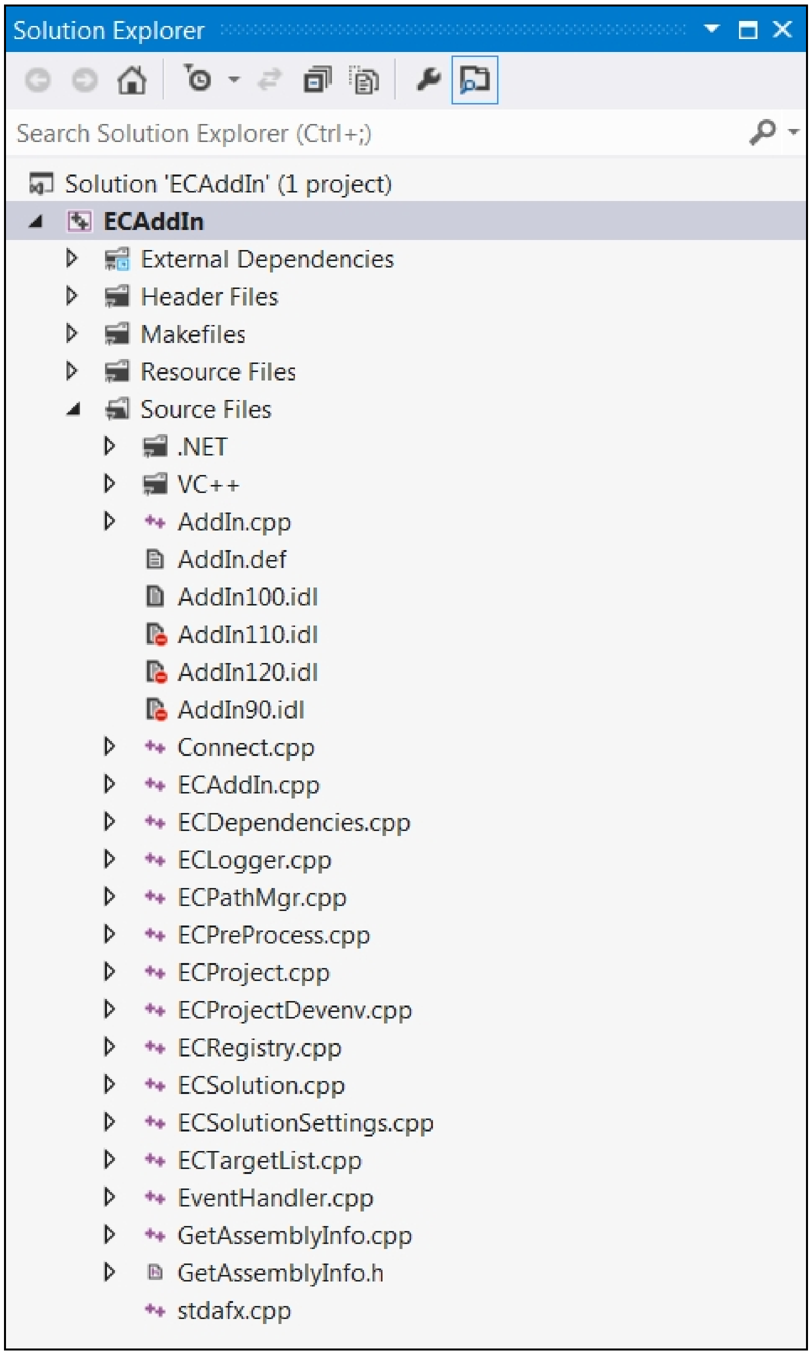
```
—-emake-jobcache=cl
```

Command Line

```
C:\ECloud\i686_win32\bin\ecdevenv.exe /build /virtualize
/configuration="ECAddIn.sln.ecaddin" /options="ECAddIn.sln.ecd" --emake-
emulation=nmake --emake-root=. --emake-annodetail=basic --emake-
localagents=y --emake-history=merge --emake-monitor=localhost:17374
```

Specifies extra EMake options

For example, if you create a simple C++ Console Application using Visual Studio, it creates two compile jobs for `Example1.cpp` and `stdafx.cpp`.



Solution 'ECAddIn' (1 project)
- ECAddIn
  - External Dependencies
  - Header Files
  - Makefiles
  - Resource Files
  - Source Files
    - .NET
    - VC++
    - AddIn.cpp
    - AddIn.def
    - AddIn100.idl
    - AddIn110.idl
    - AddIn120.idl
    - AddIn90.idl
    - Connect.cpp
    - ECAddIn.cpp
    - ECDependencies.cpp
    - ECLogger.cpp
    - ECPathMgr.cpp
    - ECPreProcess.cpp
    - ECProject.cpp
    - ECProjectDevenv.cpp
    - ECRegistry.cpp
    - ECSolution.cpp
    - ECSolutionSettings.cpp
    - ECTargetList.cpp
    - EventHandler.cpp
    - GetAssemblyInfo.cpp
    - GetAssemblyInfo.h
    - stdafx.cpp

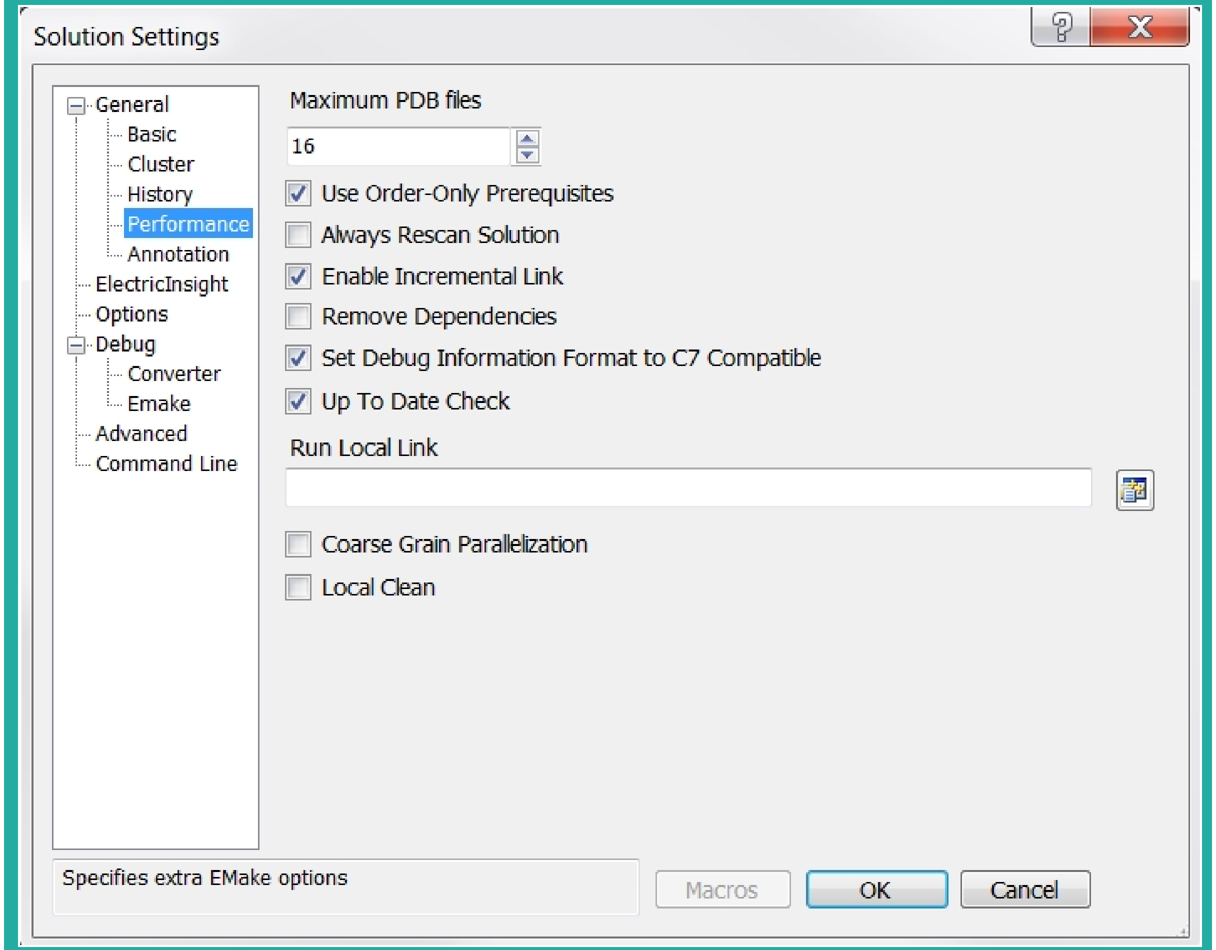
The extension converts this project into a makefile. For example:

```
".\Example1\Debug\Example1.obj":: ".\Example1\Example1.cpp"
       @cl.exe /Od /Oy- /sdl /D WIN32 /D _DEBUG /D _CONSOLE /D _LIB /D _UNICODE /D
UNICODE /EHsc /RTC1 /analyze- /MDd /GS /Zc:wchar_t /Zc:forScope /Yu"stdafx.h"
/Fp".\Example1\Debug\Example1.pch" /Fo".\Example1\Debug\Example1.obj" /W3 /WX-
/nologo /c /Z7 /Gd /errorReport:none /fp:precise /TP ".\Example1\Example1.cpp"

".\Example1\Debug\Example1.pch" :: ".\Example1\Debug\stdafx.obj"
```

```
".\Example1\Debug\stdafx.obj" :: ".\Example1\stdafx.cpp"
       @cl.exe /Od /Oy- /sdl /D WIN32 /D _DEBUG /D _CONSOLE /D _LIB /D _UNICODE /D
UNICODE /EHsc /RTC1 /analyze- /MDd /GS /Zc:wchar_t /Zc:forScope /Yc"stdafx.h"
/Fp".\Example1\Debug\Example1.pch" /Fo".\Example1\Debug\stdafx.obj" /W3 /WX-
/nologo /c /Z7 /Gd /errorReport:none /fp:precise /TP ".\Example1\stdafx.cpp"
```

When `emake` runs with JobCache, it stores `Example1.obj` and `stdafx.obj` and reuse them in the next build if it gets a "hit" (meaning that no relevant changes are detected since the last build).

IMPORTANT: JobCache does not work with the `/Zi` or `/ZI` compiler debug options when the same PDB file is updated between compilations. Each compilation must create a unique PDB file if debugging is enabled. You should use the `/Z7` option with `jobcache`. When using the Visual Studio IDE extension, check **Set Debug Information Format to C7 Compatible**:


Solution Settings

- General
  - Basic
  - Cluster
  - History
  - Performance
  - Annotation
- ElectricInsight
- Options
- Debug
  - Converter
  - Emake
- Advanced
- Command Line

Maximum PDB files: 16

- [✓] Use Order-Only Prerequisites
- [ ] Always Rescan Solution
- [✓] Enable Incremental Link
- [ ] Remove Dependencies
- [✓] Set Debug Information Format to C7 Compatible
- [✓] Up To Date Check

Run Local Link:

- [ ] Coarse Grain Parallelization
- [ ] Local Clean

Specifies extra EMake options

Troubleshooting

Enabling Basic Annotation

Basic annotation helps you to resolve your cache hit/miss problems by providing information about whether a cache miss occurred and the cause. Annotation that is related to JobCache is included in basic annotation. Basic annotation is not enabled by default, so you must do so by using the `--emake-annodetail=basic` eMake option. By default, the eMake annotation file is created in the working directory in which eMake is invoked; the file is named emake.xml by default.

Interpreting Job Cache Annotation Information

Examine the relevant `job` XML element in the annotation file for a subelement named `jobcache`.

Following is an example of a `jobcache` subelement for a job with a job cache hit. The `status` attribute indicates whether eMake used a cached object file in that job (whether it had a cache hit), and if not, what else occurred:

```
...
<job id=...>
...
     <jobcache type="gcc" options=" -exist *"
slot="6d00a0d9242610a075a98bc9400f8f11" duration="0.155831" status="hit">
...
     </jobcache>
...
</job>
...
```

The `duration` attribute is the duration (in seconds) of the job that populated the cache. If the current build is updating the cache, then the value of `duration` is the same as the duration of the job containing that `jobcache` subelement. If the current build is replaying from the cache, then it is the duration of the corresponding job from a previous build whose results were saved into the cache. eMake retrieves the figure from the cache.

Following is an example of a `jobcache` subelement for a job with a *shared* job cache hit. Note the `src="shared"` tag, which appears when Shared JobCache is used:

```
...
<job id=...>
...
  <jobcache type="gcc" options=" -exist *" slot="86e319e5cd837ad78360145fb3a933f1"
duration="0.244847" status="hit" src="shared">
    <triggers>
      <trigger type="commandline" option="--emake-jobcache=gcc"/>
    </triggers>
  </jobcache>
...
</job>
...
```

The local cache was used unless `src="shared"` appears.

Following is an example of a `jobcache` subelement for a job with a cache miss:

```
...
<job id=...>
...
     <jobcache type="gcc" options=" -exist *"
slot="989324a736c583a306630930da80ba1e" duration="0.266177" status="miss">
...       <differences>
               <diff name="/c/src/mysql-5.6.21/include/mysql.h"
old="md5:d55ae58fd7eec6055a50fcf6b83af99c"
new="md5:071ed8e989b0d0e14d0b0bd96e94cd35"/>
          </differences>
     </jobcache>
...
</job>
...
```

Following is an example of a `jobcache` subelement for a job for which caching was disabled by using `#pragma jobcache none`:

```
...
<job id=...>
...
     <jobcache type="none" options="" status="uncacheable">
          <triggers>
               <trigger type="pragma" file="Makefile" line="11" options="none"/>
          </triggers>
     </jobcache>
...
</job>
...
```

Following is a complete list of the possible values of the `status` attribute:

- `hit`—eMake had a cache hit for that job.
- `miss`—eMake had a cache miss for that job. Each `diff` subelement shows a file system input to compilation that differs since the object file was cached and shows the difference that was observed.
- `newslot`—A new slot was created. This is because the previously-cached object files were from compilations that used different command-line arguments, environment variables, working directories, or any combination of these. To see the options used for that compilation, see the "key" file for the slot identified by the `slot` attribute. For example, the key file for slot 2b253a890c9745a0b500d888349ec2e2 has the following path name:

  ```
  .emake/cache.16/i686_Linux/2b/25/3a/890c9745a0b500d888349ec2e2/key
  ```

  The version number in the cache.<*number*> directory might vary with your ElectricAccelerator software version. The key file specifies the relevant environment variables, the working directory, and the command line. To allow cache hits when building in a new workspace, path names are specified relative to your eMake roots. If a target repeatedly has `newslot` status, get the `slot` identifiers from two consecutive builds for that same target and compare the key files.
- `uncacheable`—Caching was disabled by the`#pragma jobcache none` pragma, or an error occurred during the update of the relevant cache slot. Examine any `ERROR` and `WARNING` messages in the console output from eMake.
- `rootschanged`—There is no natural mapping from the old eMake roots to the new eMake roots.

- `unneeded`—JobCache was enabled for the job but not needed (because the target was already up to date according to ordinary GNU Make rules). The cache was not consulted, even though caching was requested for that target.

If Job Cache Annotation Information Is Missing

If a particular job element in the annotation file has no XML `jobcache` subelement, this is because any combination of the following has occurred:

- The target name for that job does not match the pattern following any `jobcache` pragma, and if the target is built by an explicit rule, that rule does not follow a `jobcache` pragma.
- The intended `jobcache` pragma is misspelled.
- Either the `--emake-jobcache=` command line option was not used, or none of the targets matched
- The appropriate licensing is not available to the eMake client.

If eMake Unexpectedly Used a Cached Object File

If eMake should not have used a particular cached object file, then

- If eMake should have detected a change to a particular file, compare its path to your eMake roots.
- If a relevant environment variable changed, check whether the key file mentions it (see above), and if it does not, notify Electric Cloud.

Profiling Debug Logging

Annotation files include profiling metrics to help troubleshoot performance issues. These are the same metrics that are in the debug log file when the `--emake-debug=g` option is set. The metrics appear in annotation whether or not `--emake-debug=g` is used. The metrics are in the `<profile>` tag and appear exactly as they do in the debug log file.

Electric Cloud engineering and support staff use profiling debug logging as well as other information in the eMake debug logs to help troubleshoot problems. For more information about debug logging and log levels, see eMake Debug Log Levels on page 11-2in Chapter 11, Troubleshooting on page 11-1.

Viewing JobCache Metrics

The annotation file includes metrics about job cache activity. Following is an example that lists the metrics. This example shows that Shared JobCache is used:

```
...
<metrics>
...
     <metric name="jobcache.hit.local">382</metric>
     <metric name="jobcache.hit.shared">0</metric>
     <metric name="jobcache.hit">382</metric>
     <metric name="jobcache.miss">666</metric>
     <metric name="jobcache.newslot">98</metric>
     <metric name="jobcache.sharedmiss">0</metric>
     <metric name="jobcache.sharednewslot">0</metric>
```

```
        <metric name="jobcache.rootschanged">0</metric>
        <metric name="jobcache.uncacheable">4</metric>
        <metric name="jobcache.unneeded">6</metric>
        <metric name="jobcache.na">3150</metric>
        <metric name="jobcache.workloadsaved">367.393489</metric>
    ...
    </metrics>
    ...
```

For descriptions of these metrics, see the "Metrics in Annotation Files on page 8-6" section in Chapter 8, Annotation on page 8-1.

Moving Your Workspace

If you want to move your workspace, make sure that the new eMake roots correspond to the old eMake roots. Also, because the asset directory defaults to .emake in the current working directory, you must either copy that directory to the new workspace or use `--emake-assetdir=` to specify an asset directory that you want the two workspaces to share. If you already use `--emake-assetdir=` to point to an asset directory within your old workspace and also want to move the asset directory, you must update its value to point to the new asset directory location.

Deleting the Cache

In general, content in the cache is not deleted automatically (although it might be replaced by newer content). If the cache grows significantly beyond the size expected for a full build, you can delete the cache to save disk space.

For example, if you change the value of the `C_INCLUDE_PATH` environment variable, then the cache will grow to contain results for both the old and new values of that variable. In this case, you might want to clear the cache when you permanently change the value of this variable and therefore no longer need the old cache results.

To delete the cache, you delete the `<`*`assetdir`*`>/cache.*` directories. For example, if you are using the default asset directory on Linux, enter

```
rm -r .emake/cache.*
```

Job Caching for kati

kati is an experimental GNU make clone that is used to speed up incremental Android builds. kati is used for Android version 7 ("Nougat") to convert makefiles to Ninja files. For more information about kati, see https://github.com/google/kati.) The ElectricAccelerator JobCache feature for kati can improve build performance by caching and reusing kati target files. JobCache for Kati is supported only on Linux platforms.

JobCache caches all kati target files (.ninja files if using the Ninja build system). JobCache for kati should not be used to cache results from linking.

Because kati output target names do not follow a well-known pattern, you cannot automatically enable JobCache for kati via the `--emake-jobcache` command-line option. Instead, you must apply the

`#pragma jobcache kati` pragma inside the makefile. For example, on Android builds using kati, use the pragma as follows:

```
#pragma jobcache kati
$(KATI_BUILD_NINJA) :
```

Parse Avoidance

ElectricAccelerator includes a parse avoidance feature that can almost eliminate makefile parse time. By caching and reusing parse result files, Accelerator can speed up both full builds and incremental builds.

Parse avoidance works when emulating GNU Make with clusters or local agents.

Parse avoidance uses the concept of cache "slots." When parse avoidance is enabled, eMake maintains a slot for each combination of non-ignored command line options, non-ignored environment variable assignments, and current working directory. (Ignored arguments and environment variables are listed here.) A slot can be empty or hold a previously cached result. If the appropriate slot holds an up-to-date result, parsing is avoided.

A cached result becomes obsolete if eMake detects file system changes that might have caused a different result (with the same command line options, environment variable assignments, and current working directory). Such file system changes include any file read by the parse job, which means all makefiles, all programs invoked by $(shell) during the parse (as opposed to during rule execution), the files they read, and so on.

Console Output

When a cached parse result is reused, eMake replays the file system modifications made during the parse job and any standard output or standard error that the original parse job produced. For example, if the makefile includes

```
VARIABLE1 := $(shell echo Text > myfile)

VARIABLE2 := $(warning Warning: updated myfile)
```

then when using a cached parse result, eMake creates "myfile" and prints "Warning: updated myfile".

Enabling Parse Avoidance

You must first run a "learning" build with the parse avoidance feature enabled. To enable parse avoidance, set the following: `--emake-parse-avoidance=1`

For the learning build, (because the parse cache is empty), the argument only saves a new result to the cache. For subsequent builds, the argument enables the reuse of cached parse results and saves a new result to the cache as appropriate. If you do not specify `--emake-parse-avoidance=1`, then the parse avoidance cache is not accessed at all.

Note: You should also disable generated dependencies (either by modifying your makefiles or by using `--emake-suppress-include`; see below)

The following table describes parse avoidance-related options.

eMake Option	Description
`--emake-assetdir=<path>`	Use the specified directory for cached parse results. The default directory is named .emake. (This option also determines the location of the saved dependency information for the dependency optimization feature and the cache location for JobCache.)
`--emake-parse-avoidance=<0/1>`	Avoid parsing makefiles when prior parse results remain up-to-date and cache new parse results when appropriate.
`--emake-parse-avoidance-ignore-env=<var>`	Ignore the named environment variable when searching for applicable cached parse results. To ignore more than one variable, use this option multiple times.
`--emake-parse-avoidance-ignore-path=<path>`	Ignore this file or directory when checking whether cached parse results are up-to-date. **Append** % for prefix matching (the % must be the last character). To ignore more than one path or prefix, use this option multiple times. Incorrect placement of `%` will result in an error. Correct: `--emake-parse-avoidance-ignore-path=.foo%` Incorrect: `--emake-parse-avoidance-ignore-path=.%foo`
`--emake-suppress-include=<pattern>`	Skip matching makefile includes (such as generated dependencies). Generally, you should not suppress makefile includes unless they are generated dependency files, and you have enabled automatic dependencies as an alternative way of handling dependencies. **Note:** If the pattern does not have a directory separator, then the pattern is compared to the include's file name component only. If the pattern has a directory separator, then the pattern is taken relative to the same working directory that applies to the include directive and compared to the included file's entire path.

If a file is read during a parse but changes before eMake attempts to reuse that parse's results, the cached parse result is normally considered to be obsolete. You can, however, temporarily override this decision with `--emake-parse-avoidance-ignore-path`.

eMake permanently ignores the effect on a given parse result of certain special files that might have existed when it was created, in all cases using the names they would have had at that time:

- Anything in ".emake" or whatever alternative directory you specified using `--emake-assetdir`
- Client-side eMake debug log (as specified by `--emake-logfile`)
- Annotation file (as specified by `--emake-annofile`)
- History file (as specified by `--emake-historyfile`)

Notes:

- It is recommend that your makefiles avoid any $(shell) expansion that uses these four special files and directories in a meaningful way, because parse avoidance will ignore changes to them.
- Enabling remote parse debugging (the `--emake-rdebug` option) disables parse avoidance.

You can use a pragma to instruct parse avoidance to detect the dependence of a parse result upon path wildcard match results. This pragma makes the parse cache sensitive to the existence (or nonexistence) of specific files in directories that it reads as well as in new subdirectories thereof. (eMake does not detect matching files in just any new subdirectory, but only those that are subdirectories of directories read in the original job and their subdirectories, recursively.)

You can specify more than one glob pattern, as in the following pragma:

```
#pragma jobcache parse -readdir *.h -readdir *.c
```

For Android-specific information about parse avoidance and other Android best practices, see KB article KBEA-00130 at https://electriccloud.zendesk.com/entries/23213988-KBEA-00130-Best-practices-for-Android-builds.

Parse Avoidance Example

`noautodep.mk` is used for this example. The contents are:

```
#pragma noautodep */.git/*
$(local-intermediates-dir)/libbcc-stamp.c :

#pragma noautodep */out/target/product/generic/system/bin/cat
$(linked_module) :
```

The following parse avoidance example provides command line arguments for Android 4.1.1.

Note: You might need additional options such as `--emake-cm`

```
--emake-parse-avoidance=1 --emake-autodepend=1 --emake-suppress-include=*.d --
emake-suppress-include=*.P --emake-debug=P -f Makefile -f noautodep.mk
```

Deleting the Cache

To delete the cache, delete `<assetdir>/cache.*`. (The default asset directory is `.emake`.)

For example: `rm -r .emake/cache.*`

Moving Your Workspace

If you want to move your workspace, make sure that the new eMake roots correspond to the old eMake roots. Also, because the asset directory defaults to .emake in the current working directory, you must either copy that directory to the new workspace or use `--emake-assetdir=` to specify an asset directory that you want the two workspaces to share. If you already use `--emake-assetdir=` to point to an asset directory within your old workspace and also want to move the asset directory, you must update its value to point to the new asset directory location.

eMake looks for eMake roots in the values of Make variables. When eMake replays a cached parse result in a new workspace, it replaces the old eMake roots in the values of those variables with the new eMake roots. This policy works well as long as no confusion exists between eMake roots and other text in those variable values. For example, if the value of a Make variable is `-I/we will look into this`, your old eMake root is /we, and your new eMake root is /wg, then the new value will be `-I/wg will look into this`. For best results, choose distinctive directory names for your workspaces.

Limitations

- Windows is currently not supported.
- The parse avoidance feature does not detect the need to reparse in these cases:
  - Changes to input files and programs that are used during the parse (such as makefile includes) but are not virtualized (because they are not located under `--emake-root=...`).
  - Changes to non file system, non-registry aspects of the environment, for example, the current time of day
- If a parse job checks the existence or timestamp of a file without reading it, parse avoidance might not invalidate its cached result when the file is created, destroyed, or modified.
  - Using `--emake-autodepend=1` and `--emake-suppress-include=<`*`pattern`*`>` in conjunction with parse avoidance helps to avoid this limitation. If a generated dependency file did not exist when a parse result was saved into the cache, eMake might reuse that cached parse result even after that dependency file was created. Of course, you also benefit from eDepend's performance and reliability gains.
  - This limitation can be mitigated by the use of `#pragma jobcache parse -readdir ...`, which reruns the parse job after files are created or deleted that match the pattern and are located in a directory that was subject to a wildcard match or $(shell find ...) when the cache was populated.

Troubleshooting

Enabling Parse Avoidance Debug Logging

To help troubleshoot performance issues, you should enable parse avoidance debug logging. To do so, include a capital letter "P" in the argument of the `--emake-debug=<`*`arguments`*`>` option. Electric Cloud engineering and support staff use the eMake debug logs to help troubleshoot problems. For more information about debug logging and log levels, see the "eMake Debug Log Levels on page 11-2" section in Chapter 11,Troubleshooting on page 11-1.

Debug Log Example

You can look in "P" debug logging for an explanation of why a cached result was found to be obsolete:

```
WK01: 0.062173 Input changed: job:J08345218 slot:b6189634a793fee2fe1929fbf47cc4e4
path:/home/aeolus/t/Makefile thenSize:175 nowSize:176 ignore:0
WK01: 0.062228 Input changed: job:J08345218 slot:b6189634a793fee2fe1929fbf47cc4e4
path:/home/aeolus/t/fog.mk thenSize:0 nowSize:1 ignore:0
...
WK01: 0.062284 Cache slot is obsolete: job:J08345218
slot:b6189634a793fee2fe1929fbf47cc4e4
```

Notice that "`Makefile`" and "`fog.mk`" were both bigger; either of those changes would have triggered a reparse. The "ignore:0" comments indicate that `--emake-parse-avoidance-ignore-path` was not used to ignore the changes.

Using Key Files for Debugging

To discover why a new cache slot was used, look in "P" debug logging for the name of the "key" file for that new cache slot; for example:

```
WK01: 11.853035 Saved slot definition: .emake/cache.11/i686_
Linux/d9/0f/79/4fb975e8ff94dfa2569a303e62.new.2NPR8J/key
```

Note: The ".new.2NPR8J" portion of the slot directory name should have already been removed automatically by eMake before you need to access the key file.

Then diff that key file against other key files in sibling directories to see what parameters changed. To determine which slot directories to compare, grep for the parse job IDs in "P" debug logging. You can get parse job IDs from annotation using ElectricInsight. Each key file contains an artificial `cd` command to express the working directory, all non-ignored environment variables, and the non-ignored command-line arguments. Ignored command-line arguments and environment variables are omitted from key files. The key file is stored in a directory whose path name is derived from the md5sum of the file itself. Also, the command line might have been normalized to one that is equivalent to the original one. eMake quotes special characters according to UNIX, Linux, and Cygwin `sh` shell rules.

Ignored Arguments and Environment Variables

The following arguments do not affect which cache slot is chosen:

```
--emake-annodetail
--emake-annofile
--emake-annoupload
--emake-assetdir
--emake-big-file-size
--emake-build-label
--emake-class
--emake-cluster-timeout
--emake-cm
--emake-debug
--emake-hide-warning
--emake-history
--emake-history-force
--emake-historyfile
--emake-idle-time
--emake-job-limit
--emake-ledger
--emake-ledgerfile
--emake-localagents
--emake-logfile
--emake-logfile-mode
--emake-maxagents
--emake-mem-limit
--emake-parse-avoidance-ignore-path
--emake-pedantic
--emake-priority
--emake-readdir-conflicts
--emake-resource
--emake-showinfo
--emake-tmpdir
```

The following default set of environment variables do not affect which cache slot is chosen. You can specify additional environment variables with `--emake-parse-avoidance-ignore-env`. You must use `--emake-parse-avoidance-ignore-env` once for each variable to ignore.

```
_
BUILD_DISPLAY_NAME
BUILD_ID
BUILD_NUMBER
BUILD_TAG
CI_JENKINS_BUILD_ID
CI_JENKINS_BUILD_NUMBER
CI_JENKINS_BUILD_TAG
CI_JENKINS_HUDSON_SERVER_COOKIE
CI_JENKINS_JENKINS_SERVER_COOKIE
COLORFGBG
COLUMNS
COMMANDER_JOBID
COMMANDER_JOBSTEPID
COMMANDER_RESOURCENAME
COMMANDER_SESSIONID
COMMANDER_WORKSPACE
COMMANDER_WORKSPACE_UNIX
COMMANDER_WORKSPACE_WINDRIVE
COMMANDER_WORKSPACE_WINUNC
DBUS_SESSION_BUS_ADDRESS
DESKTOP_SESSION
DISPLAY
EMAKE_APP_VERSION
EMAKE_BUILD_MODE
EMAKE_LEDGER_CONTEXT
EMAKE_MAKE_IDEMAKE_PARSEFILE
EMAKE_PARSEFILE
EMAKE_RLOGFILE
ECLOUD_AGENT_NUM
ECLOUD_BUILD_CLASS
ECLOUD_BUILD_COUNTER
ECLOUD_BUILD_ID
ECLOUD_BUILD_TAG
ECLOUD_BUILD_TYPE
ECLOUD_RECURSIVE_COMMAND_FILE
GPG_AGENT_INFO
HOSTNAME
KDE_FULL_SESSION
KDE_MULTIHEAD
KDE_SESSION_UID
KDE_SESSION_VERSION
KONSOLE_DBUS_SERVICE
KONSOLE_DBUS_SESSION
LINES
LS_COLORS
OLDPWD
SESSION_MANAGER
SHELL_SESSION_ID
SHLVL
SSH_AGENT_PID
SSH_AUTH_SOCK
SSH_CLIENT
SSH_CONNECTION
SSH_TTY
TEMP
TERM
TERMCAP
TMP
TMPDIR
WINDOWID
WINDOWPATH
WRAPPER_ARCH
WRAPPER_BIN_DIR
WRAPPER_BITS
WRAPPER_CONF_DIR
WRAPPER_DESCRIPTION
WRAPPER_DISPLAYNAME
WRAPPER_EVENT_NAME
WRAPPER_EVENT_WRAPPER_PID
WRAPPER_FILE_SEPARATOR
WRAPPER_HOSTNAME
WRAPPER_INIT_DIR
WRAPPER_JAVA_HOME
WRAPPER_EVENT_JVM_ID
WRAPPER_EVENT_JVM_PID
WRAPPER_LANG
WRAPPER_NAME
WRAPPER_OS
WRAPPER_PATH_SEPARATOR
WRAPPER_PID
WRAPPER_SYSMEM_PP.P
WRAPPER_WORKING_DIR
XCURSOR_THEME
XDG_SESSION_COOKIE
XDM_MANAGED
```

```
WRAPPER_ARCH
WRAPPER_BIN_DIR
WRAPPER_BITS
WRAPPER_CONF_DIR
WRAPPER_DESCRIPTION
WRAPPER_DISPLAYNAME
WRAPPER_EVENT_NAME
WRAPPER_EVENT_WRAPPER_PID
WRAPPER_FILE_SEPARATOR
WRAPPER_HOSTNAME
WRAPPER_INIT_DIR
WRAPPER_JAVA_HOME
WRAPPER_EVENT_JVM_ID
WRAPPER_EVENT_JVM_PID
WRAPPER_LANG
WRAPPER_NAME
WRAPPER_OS
WRAPPER_PATH_SEPARATOR
WRAPPER_PID
WRAPPER_SYSMEM_PP.P
WRAPPER_WORKING_DIR
```

Javadoc Caching

The ElectricAccelerator Javadoc caching feature can improve build performance by caching and reusing Javadoc files.

Enabling Javadoc Caching

To enable Javadoc caching for a specific rule, add the following before that rule:

```
#pragma jobcache javadoc
```

Alternatively, you can mention the target again in a separate makefile:

```
#pragma jobcache javadoc
mytarget :
```

By default, the cache becomes obsolete when `*.java` files are added to or removed from directories that are read by the rule body. You can override this default wildcard pattern by invoking:

```
#pragma jobcache javadoc -readdir pattern
```

pattern is the pattern that you want to ignore. You can use multiple wildcard (glob) patterns.

eMake permanently ignores the effect on a given result of certain special files that might have existed when it was created, in all cases using the names they would have had at that time:

- Anything in ".emake" or whatever alternative directory you specified using `--emake-assetdir`
- Client-side eMake debug log (as specified by `--emake-logfile`)
- Annotation file (as specified by `--emake-annofile`)
- History file (as specified by `--emake-historyfile`)

Note: -hdf arguments are removed from consideration for which cache slot to use.

If you want to delete the cache, see Deleting the Cache on page 6-19.

For debugging information, see Troubleshooting on page 6-20.

Limitations

- The following Javadoc invocations are not supported:
  - Javadoc is invoked indirectly, such as through a wrapper script
  - The rule body recursively invokes eMake
  - If Javadoc is named something other than "javadoc"
- Javadoc caching is sensitive to changes in `*.java` wildcard outcomes in directories that are actually read by the job, and any new subdirectories thereof.
- `#pragma jobcache ...` cannot be applied to patterns as such, only to explicitly specified targets (though the commands might be provided by a pattern).
- Windows is not supported.
- The Javadoc caching feature does not detect the need to rebuild Javadoc files in these cases:
  - Changes to input files and programs that are used by a rule body but that Accelerator does not virtualize (because they are not located under `--emake-root=...`).
  - Changes to non file system, non-registry aspects of the environment, for example, the current time of day
- If a rule job checks the existence or timestamp of a file without reading it, Javadoc caching might not invalidate its cached result when the file is created, destroyed, or modified. However, Javadoc caching will detect when files matching "*.java" (or the pattern specified by "-readdir") are added or removed from directories read by the rule body, or any new subdirectories thereof. In that case eMake will rerun Javadoc.

Schedule Optimization

How it Works

Schedule optimization uses performance and dependency information from previous builds to optimize the runtime order of jobs in subsequent builds.

This approach can improve performance on the same number of agents (compared to v7.0, or compared to v7.1 when the requisite information is not available), and enables eMake to realize the "best possible" build performance for a given build with fewer agents.

Using Schedule Optimization

Schedule optimization is enabled by default, but it has no effect if scheduling data files are not available.

Emake automatically generates the scheduling data file in the asset directory (`.emake` by default) in the `sched` subdirectory under a platform-specific directory such as `Linux`, `SunOS`, or `win32`. For example, on Linux the scheduling data file is found in `.emake/sched/Linux/emake.sched`. The next eMake build that uses the scheduling data file will have improved performance.

You can use the `--emake-assetdir` eMake option to change the location of the asset directory.

Disabling Schedule Optimization

To disable schedule optimization, add the following to the eMake command line:

```
--emake-optimize-schedule=0
```

Running a Local Job on the Make Machine

ElectricAccelerator normally executes all build commands on the cluster hosts. Any file access by commands is served by the in-memory EFS cache or fetched from eMake and monitored to ensure that dependency information is captured to guarantee correct results.

Sometimes running a command on an Agent (which could be in the cluster or local) is not preferred. For example, a command could exercise heavy file I/O, which would create overhead that degrades performance significantly. In these instances, you can use the `#pragma runlocal` directive in a makefile to run it locally on the host build machine (where no EFS exists to virtualize and monitor file access) instead of remotely.

IMPORTANT: After a "run-local" job, eMake reloads the file system state for the current working directory of the Make, not the job.

Local jobs use "forced-serial" mode. This means that eMake executes a run-local job when all previous commands are complete. Then it does not let other commands run until the run-local job is complete. In other words, while the run-local job is running, no other build steps are executed.

Jobs That Are Suited to Running Locally

Running locally is intended only for jobs that:

- Perform heavy I/O (these jobs are inefficient to run remotely)
- Execute near the end of the build (for example, forcing the build into serial mode would have minimal performance impact)
- Logically produce a final output not used by steps later in the build

Therefore, final linking or packaging steps are typically the only commands that you should run locally.

Running locally is not intended for jobs that change the registry in a way that later parts of the build depend on. This is because these changes would not be visible to later jobs in the build.

Specifying Jobs to Run Locally

To mark a job run locally, precede its rule in the makefile with `#pragma runlocal`. For example:

```
#pragma runlocal
myexecutable:
        $(CC) -o $@ $(OBJS) ...
```

You can specify `#pragma runlocal` for a pattern rule. In this case, all jobs instantiated from that rule are marked as run-local:

```
#pragma runlocal
%.exe: %.obj
        $(CC) -o $@ $*
```

You can run jobs locally after a certain point in the build by using `#pragma runlocal sticky`. This variant specifies that all jobs later in the serial order than the job specifically marked with `#pragma runlocal sticky` should be run locally as if `#pragma runlocal` applied to all of them. This is *not* the same as "all jobs declared after this job in the makefile." The order of job declaration in a makefile is not related to the serial order of those jobs.

Making eMake Detect Files Outside the Current Working Directory

By default, eMake cannot detect files that were created by a run-local job. Instead, it attempts to find any new files in the current working directory after the job has run. This means that by default, the run-local job must not make changes outside of its current working directory. This is because if changes occur, subsequent jobs will not see the output outside of the current working directory and could fail because precise interaction depends on the file system state when the job begins.

Sometimes, however, jobs create files outside of the current working directory. If this is the case, you must use the `-repopulate` option to `#runlocal` to tell eMake where to find them instead. This is critical if later jobs that run on agents must access the output of the run-local job. You can use the `-repopulate` option to tell eMake to look in one or more directories.

The `-repopulate` option works only when `#runlocal` is specified inside an eMake pragma addendum file. For details about pragma addendum files, see the "Specifying Pragmas in an Addendum File on page 4-13" section in the "Additional eMake Settings and Features" chapter.

The syntax for using the `-repopulate` option is:

```
#pragma runlocal -repopulate <dir1> [... -repopulate <dirN>]
```

For example:

```
#pragma runlocal -repopulate ./out -repopulate ./abc/def sticky
```

Serializing All Make Instance Jobs

Normally, eMake runs all jobs in a Make instance in parallel on multiple distinct Agents. For most builds, this process ensures the best possible performance.

In some cases, however, all jobs in a Make instance are interdependent and must be run serially, for example, a set of jobs updating a shared file. In particular, Microsoft Visual C/C++ compilers exhibit this behavior when they create a common program database (PDB) file to store symbol and debug information to all object files.

Example

The following makefile invokes `cl` with the `/Zi` flag to specify the program database file be created with type and symbolic debugging information:

```
all:
       $(MAKE) my.exe
       $(MAKE) other.exe

my.exe: a.obj b.obj c.obj my.obj
               cl /nologo /Zi $^ /Fe'my.exe'

%.obj: %.c

               cl /nologo /Zi /c /Fo$@ $^
```

In this build, the `a.obj, b.obj, c.obj` and `my.obj` jobs are implicitly serialized because they all write to the shared PDB file (by default, `vc70.pdb`). In this case, jobs run in parallel, and running them on separate Agents only introduces unnecessary network overhead. This job type needs to run serially so it can correctly update the PDB file.

The eMake special directive, `#pragma allserial` used in the makefile, allows you to disable a parallel build in a Make instance and run the job serially on a single Agent. By inserting the `#pragma allserial` directive at the beginning of a line anywhere in the makefile, the directive specifies that all jobs in that make instance be serialized. This process maximizes network and file cache efficiency.

In the example above, by prefixing the `%.obj` pattern rule with the `#pragma allserial` directive:

```
#pragma allserial
%.obj: %.c
        cl /nologo /Zi /c /Fo$@ $^
```

eMake runs compiles and links for the `my.exe` Make instance in serial on the same Agent.

Splitting PDBs Using hashstr.exe

The `hashstr.exe` utility creates a hash of the file name given a modulus (maximum number of PDBs that will be produced). A given file must always produce the same PDB or history would constantly change. The hash should only include the file name and not its full path. Precompiled headers (PCHs) must be turned off.

Example

Usage: `hashstr "mystring" [modulus]`

Where mystring is the string from which to generate the hash value, and modulus is the number of hash bins you want to use.

You can add this to a pattern rule for builds that suffer from performance degradation due to PDB serialization, with something similar to the following:

```
%.o: %.c
$(CC) /c $(cflags) $(PCH_USE_FLAGS) $(cvars) $(cplus_flags) $(LOCAL_INCLUDE)
$(PCB_INCLUDE) $< /Fo$@ /Fd$(shell ${path-to-hashstr}/hashstr.exe "$@"
${hashstr-modulus}).pdb
```

Managing Temporary Files

During the build process, temporary files are created, then deleted automatically when a build completes. To maintain optimum efficiency, it is important to identify where these files are created in relation to the final output files.

Configuring the eMake Temporary Directory

eMake runs commands and collects output in parallel, but the results are written in order, serially. This feature ensures a build behaves exactly as if it were run locally. See Transactional Command Output on page 5-4.

The eMake temporary directory is used to store output files from commands (for example, object files) that finished running but are waiting their turn to be relocated to a final destination on disk. By default, eMake creates a temporary directory in the current working directory. The directory name will have the form:

`ecloud_tmp_<pid>_<n>`
where pid is the eMake process identifier.
where n is a counter assigned by eMake to differentiate multiple temporary directories created during a single build if you specify multiple directories.

eMake removes the temporary directory on exit (including the `Ctrl-C` user interrupt). If eMake is terminated unexpectedly (for example, by the operating system), the temporary directory might persist and need to be removed manually.

Important temporary directory requirements include:

- Must be writable by the eMake process
- Must not be on an NFS share
- Must have enough space to contain the build output
    - Ideally, the temporary directory should have enough space to hold the complete build output; in practice however, it might need only enough space for output from the largest commands.
    - The exact temporary directory space requirement varies greatly with the build size, the number of Agents used, and build system speed.

You can use the `--emake-tmpdir` command-line option or the `EMAKE_TMPDIR` environment variable to change the temporary directory default location:

```
% emake --emake-tmpdir=/var/tmp ...
```

Files are relocated from the temporary directory to their final location. Keeping the temporary directory on the same file system as the `EMAKE_ROOT` helps performance because files can simply be renamed in place, instead of copied.

eMake and agent machines reset the values for TEMP, TMP, and TMPDIR. This is necessary to avoid possible conflicts with multiple build agents/jobs running on the same machine.

If you have more than one `EMAKE_ROOT` that spans multiple file systems, you can specify more than one temporary directory to ensure eMake can always rename files in place, instead of copying them.

For example, if your `EMAKE_ROOT` contains three directories:

```
% setenv EMAKE_ROOT /home/alice:/local/output:/local/libs
```

that reside on two different file systems

```
% df /home/alice /local
Filesystem      1K-blocks  Used     Available Use% Mounted on
filer:/vol/     76376484 47234436 25262352    66%   /home
vol0/space
/dev/hdb1       76896316 51957460 21032656     2%   /local
```

to ensure better performance, specify two temporary directories—write permission at both locations is required:

```
% emake --emake-tmpdir=/home/alice:/local ...
```

When specifying multiple temporary directories, note the following:

- If a temporary directory was specified for a particular file system, eMake automatically uses that directory for any files destined to reside on that file system.
- If a temporary directory was not specified for a particular file system, eMake uses the *first* directory specified for files destined to reside on that file system.

Deleting Temporary Files

During a build, eMake creates a temporary directory inside the directory specified by the `EMAKE_TMPDIR` environment variable or in the directory specified by the `--emake-tmpdir` command-line option (or the current working directory if no eMake temporary directory is specified). If a specified directory does not exist, eMake creates it. All temporary directories created by eMake are automatically deleted when a build completes.

For example, if you set your temporary directory to `/foo/bar/baz` and only `/foo/bar` exists, eMake creates `/foo/bar/baz` and `/foo/bar/baz/ecloud_tmp_<pid>_<n>`, and then deletes both directories and all their contents when it exits. If `/foo/bar/baz` exists at the start of the build, only `/foo/bar/baz/ecloud_tmp_<pid>_<n>` is created and deleted.

However, if the build is aborted in-progress, eMake will not have the opportunity to remove the temporary directory.

Subsequent eMake invocations automatically delete temporary directories if they are more than 24 hours old. You can reclaim disk space more quickly by deleting temporary directories and their contents by hand. However, *do not delete the temporary directory while a build is in-progress*—this action causes the build to fail.

Temporary directory names are in this form:

`ecloud_tmp_<pid>_<n>`
where pid is the eMake process identifier.

where n is a counter assigned by eMake to differentiate multiple temporary directories created during a single build if you specify multiple directories.

Chapter 7: Dependency Management

The following topics discuss ElectricAccelerator eDepend, the Ledger, and how ElectricAccelerator handles history data files.

Topics:

- ElectricAccelerator eDepend
- ElectricAccelerator Ledger File
- Managing the History Data File

ElectricAccelerator eDepend

In its default configuration, Accelerator is designed to be a drop-in replacement for your existing Make tool—GNU Make or Microsoft NMAKE. Accelerator behaves exactly like your existing Make: it will rebuild (or declare up-to-date) the same targets following the same rules Make uses. The file dependency tracking technology in the Electric File System (EFS) and the eMake history feature is used to ensure the system reproduces exactly the same results in a parallel, distributed build as it would serially.

Because the system captures and records such detailed information about the relationships between build steps, it is uniquely capable of accomplishing much more than simply ensuring parallel builds are serially correct. In particular, by enabling ElectricAccelerator eDepend, you can wholly replace tools and techniques like **makedepend** or **gcc -M**—commonly used to generate makefiles too difficult to maintain by hand (for example, C-file header dependencies).

ElectricAccelerator eDepend is easier to configure, faster, more accurate, and applicable to a much wider range of dependencies than existing dependency generation solutions. If your current build does not employ dependency generation, you can enable eDepend and benefit from more accurate incremental builds without the overhead of configuring and integrating an external dependency generation tool.

The following sections describe the dependency generation challenge in more detail and how eDepend can improve your build speed and accuracy.

Dependency Generation

Consider a build tree that looks like this:

`src/`

`Makefile`	<--- top-level makefile: recurses into `mylib` and then into `main` to build the program
`common/`	
`header1.h`	
`header2.h`	
`mylib/`	
`Makefile`	<--- has rules to build `mylib.o` and create library `mylib.a`
`mylib.h`	
`mylib.c`	<--- includes `common/header1.h` and `mylib.h`
`main/`	
`Makefile`	<--- has rules to build `main.o` and then to link `main` using `main.o` and `mylib.a`
`main.c`	<--- includes `common/header1.h`, `common/header2.h`, and `lib/mylib.h`

The Problem

Even in this simple example, the need for dependency generation is apparent: if you make a change to a header file, how do you ensure dependent objects are recompiled when you rebuild?

Makefiles could explicitly declare all header dependencies, but that quickly becomes too cumbersome: each change to an individual source file might or might not require an adjustment in the makefile. Worse, conditionally compiled code can create so many permutations that the problem becomes intractable.

Another possibility is to declare all headers as dependencies unilaterally, but then the build system becomes very inefficient: after a successful build, a modification to `header2.h` should trigger a rebuild only of the `main` module, not `mylib.a` as well.

Clearly, to get accurate, efficient builds, the system must have calculated dependencies automatically *before* it builds.

There are several ways to generate dependencies and update makefiles to reflect these dependencies (for example, **makedepend** or **gcc -M**), but they all have the drawbacks mentioned previously.

eDepend Benefits

ElectricAccelerator eDepend is an eMake feature that directly addresses all problems with existing dependency generation solutions. Specifically:

- It is part of eMake and requires no external tool to configure, no extra processing time, and it is faster than other solutions.
- It is easily enabled by setting a command-line parameter to eMake. No tools or changes to makefiles are required.
- Like ElectricAccelerator itself, it is completely tool and language independent. eDepend automatically records any and all kinds of dependencies, including implicit relationships such as executables on libraries during a link step.
- eDepend dependencies are recorded in eMake history files—transparently recorded and used without manifesting as makefile rules.
- eDepend is accurate because it uses file information discovered by the Electric File System at the kernel level as a job executes its commands.

How Does eDepend Work?

Internally, eDepend is a simple application of sophisticated file usage information returned by the Electric File System.

1. As a job runs, the Electric File System records all file names it accesses inside `EMAKE_ROOT`.

   This function has two very important implications:

   - eDepend can track dependencies within `EMAKE_ROOT` only.
   - eDepend can track dependencies for a job only after it has run—this is why you must start with a complete build rather than an incremental build.

2. After a job completes, eMake saves the following eDepend information to the eMake history file:
   - the working directory for the Make instance
   - the target name
   - any files actually read (not just checked for existence) and/or explicitly listed as prerequisites in the makefile

   Note: eDepend information is stored in the history file along with serialization history information. Commands operating on the history file (for example, those specifying file location or that erase it) apply to eDepend information as well.

3. In a subsequent build, whenever eMake schedules a target in a directory for which it has eDepend information, it evaluates file dependencies recorded in the earlier run as it checks to see if the target is up-to-date.

   In the example above, the rule to update `mylib.o` might look like this:

   ```
   mylib.o: mylib.c
          $(CC) ...
   ```

`mylib.c` includes `common/header1.h`, which is not explicitly listed as a prerequisite of `mylib.o`, so eDepend records this implicit dependency in the history file.

Directory	Object	Dependency
src/mylib	mylib.o	common/header1.h

If a change is then made to `common/header1.h`, `src/mylib/mylib.o` it will be rebuilt.

Enabling eDepend

1. Start from a clean (unbuilt) source tree.
2. If your build system already has a dependency generation mechanism, turn it off if possible. If you cannot turn it off, you will still get eDepend's additional accuracy, but you will not be able to improve the performance or shortcomings of your existing system.
3. Build your whole source tree with eDepend enabled.

   Use the `--emake-autodepend=1` command-line switch:

   ```
   % emake --emake-autodepend=1 ...
   ```

   Alternatively, insert `--emake-autodepend=1` into the `EMAKEFLAGS` environment variable.

   ```
   % setenv EMAKEFLAGS --emake-autodepend=1
   % emake ...
   ```

4. Make a change to a file consumed by the build, but not listed explicitly in the makefiles.

   For example, touch a header file: `% touch someheader.h`

5. Rebuild, again making sure eDepend is enabled.

   ```
   % emake --emake-autodepend=1 ...
   ```

Notice that without invoking a dependency generator, eMake detected the changed header and rebuilt accordingly.

Important Notes

- The eDepend list is consulted only if all other prerequisites in the makefile indicate the target is up-to-date.

  Explained another way: If a target is out-of-date because it does not exist or because it is older than one of the prerequisites listed in the makefile, eDepend costs nothing and has no effect.

  If the target is newer than all its listed prerequisites, then eDepend is the "11th hour" check to ensure it really is up-to-date, and that there is not a newer implicit dependency. This is the only place eDepend interacts with your build: it forces a target that incorrectly appears to be up-to-date to be rebuilt.

- eDepend information, unlike traditional Makedepend rules, does not in any way imply anything about needing to build or update the implicit prerequisite.

  In the example above, if `header1.h` is renamed or moved, eMake just ignores the stale eDepend information. When eMake next updates the `mylib.o` target, it will prune stale dependencies from the eDepend list. This change to the history file occurs regardless of the setting of the `--emake-history-force` parameter to eMake.

  Unlike Make, eMake does not complain if it does not have a rule to make `header1.h` because eDepend dependencies are not used to schedule targets.

- eMake considers a rule to be out of date when an implicit dependency is removed. This causes eMake to rebuild the target, matching ClearMake.

- eDepend's information scope is bound by a directory name and the target name. This means you can build cleanly from the top of a tree, then run accurate incremental builds from individual subdirectories and eDepend information will be used and updated correctly.

  However, it does imply if you have a build that

  - performs multiple passes or variants over the same directories
  - with exactly the same target names, but
  - runs significantly different commands

  For example, a build that produces objects with the same name for a different architecture or configuration, eDepend information might be over-used unnecessarily. In this case, eMake might rebuild more than is necessary, but with no incorrect build results. In this situation, you can achieve fast, correct builds by using separate history files, or ideally, by changing to unique target names across build variants.

Using #pragma noautodep

Some build steps contain many implicit dependencies that might not make sense to check for up-to-dateness. Examples include symbol files consumed by a link step or archive packager input files (for example, *tar*). In both cases, any makefile explicit prerequisites are sufficient to determine if the target should be updated: eDepend information would just add overhead or cause unnecessary rebuilds.

You can selectively disable eDepend information for certain files from any step by supplying eMake with the makefile directive:

```
#pragma noautodep *.pdb
%.o: %.c
        $(CL) ...
```

The directive `#pragma noautodep` is applied to the next rule or pattern rule in the makefile. This directive specifies a class of files for eDepend to ignore. Note the following information about `#pragma autodep`:

1. Wildcards allowed for `#pragma autodep`:

   * matches 0 or more characters

   ? matches 1 character

   [ ] matches a range of characters

2. The `noautodep` filters are matched against absolute path names. To limit a filter to files in the current directory for the job, use '`./`':

   ```
   #pragma noautodep ./foo.h
   ```

   To specify "ignore foo.h" in any directory, use:

   ```
   #pragma noautodep */foo.h
   ```

3. If the supplied pattern has no wildcards and does not specify a path, it will never match.

   eMake ignores the directive and prints a warning as it parses the makefile:

   ```
   Makefile:2: ignoring autodep filter 'foo',
   does not match absolute path(s).
   ```

ElectricAccelerator Ledger File

Traditional Make facilities rely exclusively on a comparison of file system timestamps to determine if the target is up-to-date. More specifically, an existing target is considered out-of-date if its inputs have a "last-modified" timestamp later than the target output.

For typical interactive development, this scheme is adequate: As a developer makes changes to source files, their modification timestamps are updated, which signals Make that dependent targets must be rebuilt. There is, however, a class of workflow styles that cause file timestamps to move arbitrarily into the past or future, and therefore circumvent Make's ability to correctly rebuild targets.

Two common examples are:

- Using a version control system that preserves timestamps on checkout (also known as "sync" or "update").

The default mode for most source control systems is to set the last-modified timestamp of every file updated in a checkout or sync operation to the current day and time. If you change this behavior to preserve timestamps (or if your tool's default mode is *preserve*), then updating your source files can result in modified contents but with a timestamp in the past (typically, it is the time of the checkin).

- Using file or directory synchronization tools (even simple recursive directory copies) to keep files updated against some other repository.

  Here again, while it is easy to modify source file content, the timestamp for modifications might be any of several possibilities: time of copy, last-modified time of source, last-modified time of destination, and so on.

The Problem

In all modified source files cases, we would like the Make system to rebuild any dependent objects. However, because timestamps of modified files are not set reliably, Make might or might not force a target update. Here is an example Makefile:

```
foo.o: foo.c
        gcc -c foo.c

foo.o: foo.h
```

And a build is run without an existing `foo.o` object:

```
% make
gcc -c foo.c

% ls -lt
total 4
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.o
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.c
-rw-r--r-- 1 jdoe None 20 Apr 25 17:34 foo.h
-rw-r--r-- 1 jdoe None 41 Jan 19 09:27 Makefile
```

The `foo.o` target is updated. Next, suppose we ask our source control system to update the working directory, and it responds by giving us a newer copy of `foo.h`, one that is several weeks newer than what we have, and *that* timestamp is preserved:

```
% <sync>
% ls -lt
total 4
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.o
-rw-r--r-- 1 jdoe None 21 May 29 13:50 foo.c
-rw-r--r-- 1 jdoe None 29 May 17 11:21 foo.h <-- notice timestamp change
-rw-r--r-- 1 jdoe None 41 Jan 19 09:27 Makefile
```

Traditional Make programs (here, GNU Make) will not notice the change because the timestamp is still in the past, and will incorrectly report that the target is up-to-date.

```
% make
make: `foo.o' is up to date.
```

Some Make facilities (notably, Rational 'clearmake' in conjunction with Rational ClearCase) have the ability to track timestamp information because they are integrated with the source control system.

The eMake Solution

eMake solves this problem at the file level, completely independent of the source control system, by keeping a separate database of inputs and outputs called a ***ledger***. To use the Ledger, you specify which file aspects to check for changes when considering a rebuild. To do so, use the `--emake-ledger=<valuelist>` command-line switch (or the `EMAKE_LEDGER` environment variable). `<valuelist>` is a comma-separated list that includes one or more of: `timestamp`, `size`, `command`, `nobackup`, `nonlocal`, and `unknown`. For more information, see Ledger options in eMake Command-Line Options, Environment Variables, and Configuration File on page 3-8.

- `timestamp` – Any timestamp changes to either the target or the explicitly declared dependency, regardless of how it relates to the last modified time of the target input file, triggers a target rebuild.
- `size` – Any size change, regardless of the timestamp in the input file, triggers a target rebuild.
- `command` – Records the text of the command used to create the target. If makefile or its variables change, using `command` rebuilds the target. **Important caveat:** If you initialize a variable using the $(*shell*) function, be extremely careful to use the $(*shell*) function with a ':=' assignment to avoid re-evaluating it every time the variable is referenced. ':=' simply expanded variables are expanded immediately upon reading the line.
- `nobackup` – Suppresses the automatic backup of the ledger file before its use.
- `nonlocal` – Instructs eMake to operate on the ledger file in its current location, even if it is on a network volume. By default, if the file specified by `--emake-ledgerfile` (emake.ledger in the current working directory, by default) is not on a local disk, eMake copies that file (if it already exists) to the system temporary directory and opens the copy, then copies it back to the specified location when the build is complete.

  Using `nonlocal` removes a safety and might cause problems if the non-local file system has issues with memory-mapped I/O (IBM Rational ClearCase MVFS is known to have issues with memory-mapped I/O). If you are confident that you will get efficient and reliable memory-mapped I/O performance from the non-local file system, you can remove the safety for improved efficiency because eMake does not spend time at startup and shutdown copying ledger files. Electric Cloud strongly recommends against using `nonlocal` with ClearCase dynamic views. Electric Cloud does not support Ledger-related problems that occur when `nonlocal` is used in conjunction with the MVFS.
- `unknown` – Specifies that the Ledger feature consider a target to be out of date, if the Ledger database contains no entry for the target.

In the example above, the Ledger can detect if a rebuild is necessary as the timestamps change. If the original build was:

```
% emake --emake-ledger=timestamp
gcc -c foo.c
% <sync> <-- notice timestamp change
% emake --emake-ledger=timestamp
gcc -c foo.c
```

eMake consulted the Ledger and concluded the target needed to be rebuilt.

Important Notes for the Ledger Feature

- The Ledger feature works by comparing an earlier input state with the current state: if the Ledger has no information about a particular input (for example, during the first build after it was added to a makefile), it will not contribute in the up-to-dateness check.
- Only one Ledger is used per build.
- The default ledger file is called `emake.ledger`

  It can be adjusted by the `--emake-ledgerfile=<path>` command-line option or `EMAKE_LEDGERFILE=<path>` environment variable.
- If you specify `--emake-ledgerfile=<path>` but not `--emake-ledger=<valuelist>`, the Ledger still hashes the file names, so the Ledger is triggered when the file name order changes or a file is added or removed.
- The Ledger automatically backs up the ledger file before using it. This ensures a non-corrupt file is available. If the ledger file is large, copying it could take some time on incremental builds. The ledger option, `nobackup`, suppresses the backup.
- Ledger works for local builds and those using a cluster, as well as local submakes in a runlocal job, see Running a Local Job on the Make Machine on page 6-25.
- It is not possible, however, to share a Ledger between top-level make instances and local-mode submakes running on the cluster. See EMAKE_BUILD_MODE=local in eMake Command-Line Options, Environment Variables, and Configuration File on page 3-8.
- eMake consults Ledger information to trigger a rebuild only when a target would otherwise be considered up-to-date. Information in the Ledger never prevents a target from being rebuilt.
- In a GNU Make emulation, the Ledger feature changes the meaning of the '`$?`' automatic variable to be synonymous with '`$^`' (all prerequisites, regardless of up-to-dateness).
- You cannot change Ledger options for a particular ledger file—you must use the same combination of timestamp, size, and command that was used to create the ledger file.
- If you turn on `--emake-ledger` and `--emake-autodepend` at the same time, the Ledger keeps track of both implicit and explicit dependencies. This feature is comparable to using ClearMake under ClearCase, but is independent of ClearCase information records.
- Order-only prerequisites, in keeping with their semantic meaning, never affect Ledger behavior.
- Because the Ledger automatically rebuilds a target when there is no existing entry in the ledger file, a build that is using the Ledger for the first time might take longer than expected.

Managing the History Data File

When Accelerator runs a build for the first time, it takes aggressive action to run all jobs as fast as possible in parallel. Jobs that run in the wrong order because of missing makefile dependencies are automatically re-run to ensure correct output. (These failed steps are also called *conflicts*. See Conflicts and Conflict Detection on page 7-13 for information about conflicts.)

To avoid the cost of re-running jobs on subsequent builds, eMake saves the missing dependency information in a *history data file*. The history data file evolves with each new run and enables Accelerator to run builds at peak efficiency.

You can choose the location of the history file and how it is updated.

Setting the History File Location

By default, eMake creates the history file in the directory you use to invoke the build and names it `emake.data` by default. The file location can be explicitly specified using the command-line option:

```
--emake-historyfile=<pathname>
```

The history file is used for two operations during an eMake cluster build:

- Input—eMake reads the history file as it starts a build to improve build performance.
- Output—eMake writes to the history file as it completes a build to improve performance of subsequent builds.

History File Input Rules

If the history file (`emake.data` or whatever was specified with `--emake-historyfile`) exists, it is *always* read and used to improve performance.

History File Output Rules

Data written to the history file after the build depends on the `--emake-history` option setting. Three options are available:

1. Merge—By default, eMake merges new dependencies into the existing history file. In this way, the history file evolves automatically as your makefiles change by learning dependencies that accelerate your builds.
2. Create—If `--emake-history` is set to `create`, the old history file contents are overwritten by new dependencies discovered in the run that just completed. Use this setting to start a fresh history file to eliminate stale information from the file.
3. Read—If `--emake-history` is set to `read`, no data is written to the history file at build completion, and any new dependencies discovered are discarded. Use this setting when developers share a single, static copy of the history file.

By default, the history file is updated even if the build fails, regardless of the value of `--emake-history`. You can override this behavior by setting `--emake-history-force=0`.

The history file directly impacts the number of conflicts the build can encounter. Ideally, an ElectricAccelerator cluster build with good history should have almost no conflicts. If conflicts are increasing, check for a current history file.

Guaranteeing Correct History

Use `--emake-readdir-conflicts=1` to guarantee correct history. Some parallel builds do not succeed without a good history file. In particular, builds that use wildcard or globbing operators to produce build-generated lists of files and operate on those lists might fail. For example, a makefile might use `ld *.o` as shorthand to avoid enumerating all the `*.o` files in a directory. Running the build with `--emake-readdir-conflicts=1` guarantees that the build succeeds and that a history file is created for use by subsequent parallel builds.

Do *not* enable `--emake-readdir-conflicts=1` all the time. Instead, enable it for one run if you suspect a globbing problem, and then disable it, but use the history file generated by the previous run.

You can alternatively use the `#pragma readdirconflicts` pragma to enable directory-read conflicts on a per-job basis. You can apply it to targets or rules in your makefiles. This pragma has less overhead than `--emake-readdir-conflicts=1` (which enables directory-read conflicts for an entire build). You can use this pragma in pragma addendum files as well as in standard makefiles.

Ensuring that Relative EMAKE_ROOT Locations Match

Relative `EMAKE_ROOT` locations must match. The history file records target file names relative to the `EMAKE_ROOT` specified during that run. For a subsequent build to use the history file correctly, target file names must have the same path name relative to the eMake root.

For example, if your eMake root is `/home/alice/builds` and your build references a path name in that root: `/home/alice/builds/lib/foo.o`, then the history file records it as `lib/foo.o`. If a subsequent build sets the eMake root to `/home/bob/builds`, the history file will match correctly.

If, however, the eMake root is `/home/bob`, then the file that exists on the disk as `/home/bob/builds/lib/foo.o` is assigned the root-relative name of `builds/lib/foo.o`, which does not match the name `lib/foo.o` in the history file generated above. Because the history file does not match, performance might suffer.

Note: `EMAKE_ROOT` must match the same location relative to sources as the `EMAKE_ROOT` used to create the history file.

Running Builds with Multiple Roots

For builds with multiple roots, the roots must have the same alphabetical sorting order in each build so that the history matches.

Using the remaphist Utility to Relocate a History File

The `remaphist` utility makes it easier for users to share history files. It is located at:

- (Linux) `<install_dir>/i686_Linux/unsupported/remaphist`, where `<install_dir>` is `/opt/ecloud` by default
- (Windows) `<install_dir>\i686_win32\unsupported\remaphist`, where `<install_dir>` is `C:\ECloud` by default

Modes of Operation

The `remaphist` utility has two modes of operation:

1. Makes a standard eMake history file "relocatable" (and therefore usable in other build environments).

   History files store paths in the form *<root_ID><root_relative_path>*, so if your root is `/home/stevec` and you have a path such as `/home/stevec/proj1`, the history file records it as `0 proj1`. A relocatable history file flattens those references and then replaces the root prefixes with variables that are easier to "swizzle" later. For example, this step takes `0 foo` to `/home/stevec/proj1` to `$(ROOT0)/proj1`.

2. Converts a relocatable history file back into a standard history file.

   This step expands the variables according to the new user's specification and then converts them into new root-relative paths. For example, this step takes `$(ROOT0)/proj1` to `/home/tmurphy/foo` to `0 foo`.

Syntax

To remap a standard file to make it relocatable:

```
remaphist -i <input_file> -o <mapped_file> [-r <emake_root(s)>] <VARIABLE>=<absolute_
path> [<VARIABLE2>=<absolute_path2> ... ]
```

To convert a relocatable file back to a standard file:

```
remaphist -u -i <mapped_file> -o <output_file> -r <emake_root(s)><VARIABLE>=<absolute_
path> [<VARIABLE2>=<absolute_path2> ... ]
```

Option	Description
`-u`	Performs an unmapping. The default is to perform a remapping.
`-i` *<input_file>*	Path to the unmapped (standard) input file.
`-o` *<mapped_file>*	Path to the remapped (relocatable) file to create.
`-i` *<mapped_file>*	Path to the remapped (relocatable) input file.
`-o` *<output_file>*	Path to the unmapped (standard) file to create.
`-r` *<emake_root(s)>*	Specifies the eMake root(s).
`-s` *<0|1>*	(Optional) Specifies whether to sort the root directories. The default is `1` (sorted).
<VARIABLE>=<absolute_path> [*<VARIABLE2>=<absolute_path2>* ...]	eMake roots. You can specify one or more roots.
`<-help|?>`	(Optional) Prints the help message.

Examples

The following example shows how to remap a standard history file to make it relocatable:

```
remaphist -i emake.data -o remapped.data PATHVAR=/opt/chrish/work
PATHVAR2=/opt/chrish/work2/q1proj
```

The following example shows how to convert a remapped history file back to a standard file:

```
remaphist -u -i remapped.data -o emake.data2 -r /workspace/tools
MYVAR=/opt/kathy/abs/proj MYVAR2=/opt/kathy/modules/
```

Conflicts and Conflict Detection

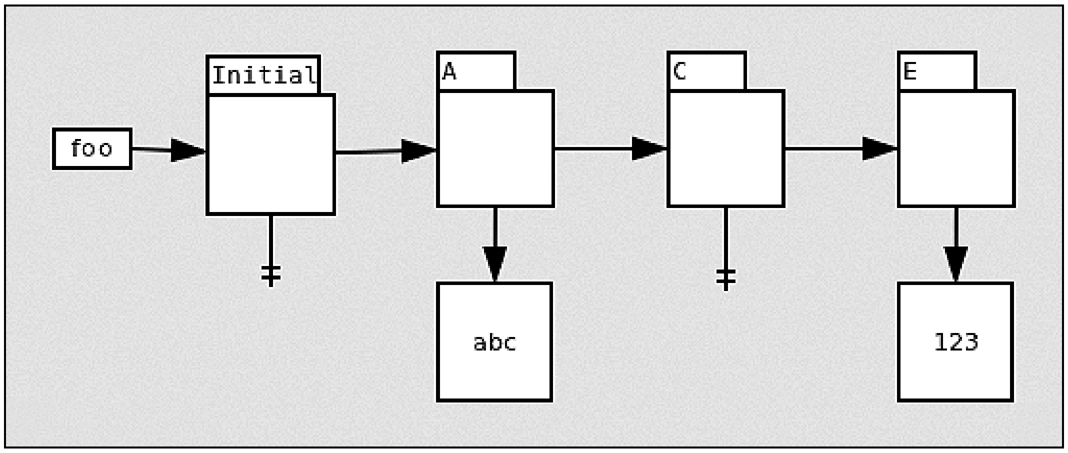
This section discusses conflicts and conflict detection in eMake. This content was adapted from the *How Electric Make guarantees reliable parallel builds* and *Exceptions to conflict detection in ElectricMake* blog posts on http://blog.melski.net/.

How eMake Guarantees Reliable Parallel Builds

The technology that enables eMake to ensure reliable parallel builds is called conflict detection. Although there are many nuances to its implementation, the concept is simple. First, track every modification to every file accessed by the build as a distinct version of the file. Then, for each job run during the build, track the files used and verify that the job accessed the same versions it would have if the build ran serially. Any mismatch is considered a conflict. The erroneous job is discarded along with any file system modifications that it made, and the job is rerun to obtain the correct result.

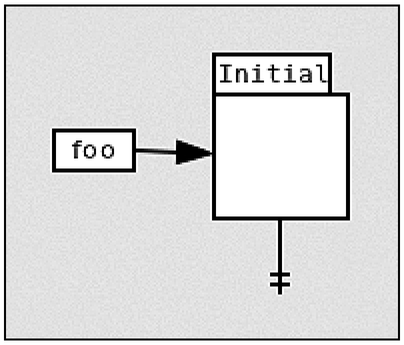
The Versioned File System

The main part of the conflict detection system is a data structure called the versioned file system, in which eMake records every version of every file used over the lifetime of the build. A version is added to the data structure every time a file is modified, whether that is a change to the content of the file, a change in the attributes (such as ownership or access permissions), or the deletion of the file. In addition to recording file state, a version records the job that created it. For example, here's what the version chain looks like for a file "foo," which initially does not exist, then is created by job A with contents "abc", deleted by job C, and recreated by job E with contents "123":

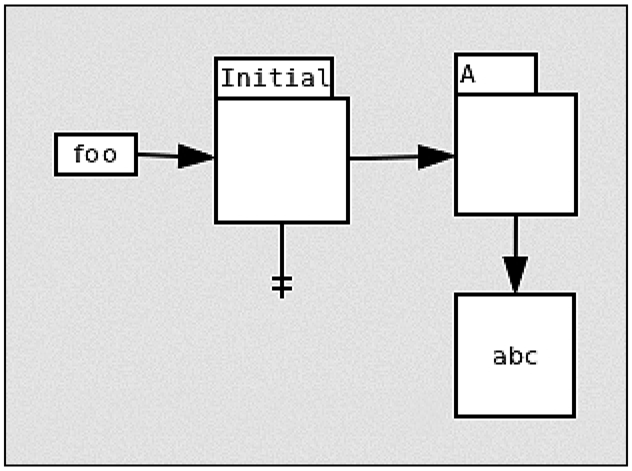


Detecting Conflicts

With all the data that eMake collects—every version of every file, and the relationship between every job—the actual conflict check is simple: For each file accessed by a job, compare the actual version to the serial version. The actual version is the version that was actually used when the job ran; the serial version is the version that would have been used, if the build had run serially. For example, consider a job B, which attempts to access a file "foo". At the time that B runs, the version chain for "foo" looks like this:



Given that state, B will use the initial version of "foo"—there is no other option. The initial version is therefore the actual version used by job B. Later, job A creates a new version of foo:



Because job A precedes job B in serial order, the version created by job A is the correct serial version for job B. Therefore, job B has a conflict.

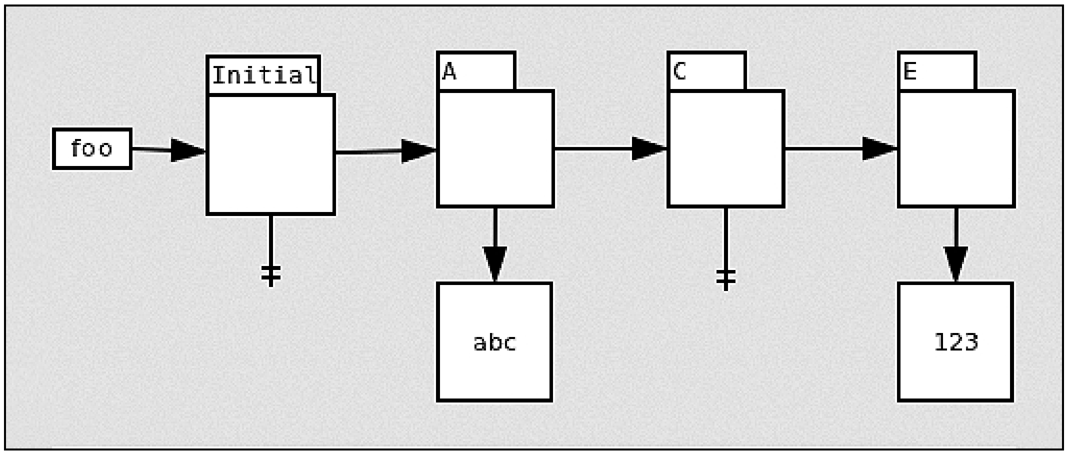
If a job is found to be free of conflicts, the job is committed, meaning that any file system updates are at last applied to the real file system. Any job with a conflict is reverted — all versions created by the job are marked invalid, so subsequent jobs will not use them. The conflict job is then rerun to generate the correct result. The rerun job is committed immediately upon completion.

Conflict checks are done by a dedicated thread that inspects each job in strict serial order. This guarantees that a job is not checked for conflicts until after every job that precedes it in serial order is successfully verified to be free of conflicts. Without this guarantee, the system cannot be sure that it knows the correct serial version for files accessed by the job. Similarly, this ensures that the rerun job, if any, uses the correct serial versions for all files, so the rerun job is sure to be conflict free.

Exceptions to Conflict Detection in eMake

Non-Existence Conflicts

An obvious enhancement is to ignore conflicts when the two versions are technically different, but effectively the same. The simplest example is when there are two versions of a file, which both indicate non-existence, such as the initial version and the version created by job C in this chain for file “foo”:



Suppose that job D, which falls between C and E in serial order, runs before any other jobs finish. At runtime, D sees the initial version, but strictly speaking, if it had run in serial order, it would have seen the version created by job C. But the two versions are functionally identical—both indicate that the file does not exist. From the perspective of the commands run in job D, there is no detectable difference in behavior regardless of which of these two versions was used. Therefore, eMake can safely ignore this conflict.

Directory Creation Conflicts

A common make idiom is `mkdir -p $(dir $@)`—that is, create the directory that will contain the output file, if it doesn’t already exist. This idiom is often used as follows:

```
$(OUTDIR)/foo.o: foo.cpp
    @mkdir -p $(dir $@)
    @g++ -o $@ $^
```

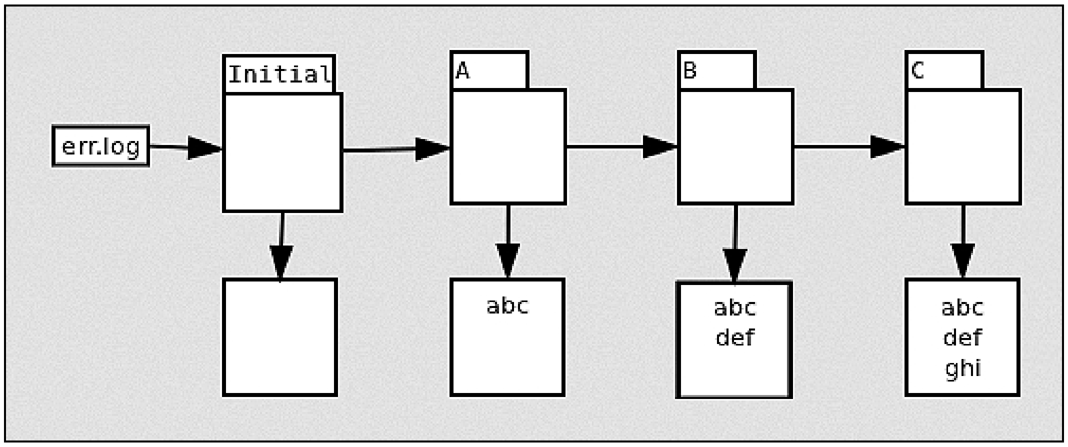
Suppose that the directory does not exist when the build starts, and several jobs that employ this idiom start at the same time. At runtime, they will each see the same file system state—namely, that the output directory does not exist. Therefore, each job will create the directory. But in reality, had these jobs run serially, only the first job would have created the directory; the others would have seen the version created by the first job, and done nothing with the directory themselves. According to the simple definition of a conflict, all but the first (serial order) job would be considered in conflict. For builds without a history file expressing the dependency between the later jobs and the first, the performance impact would be substantial.

Appending to Files

Another surprisingly-common idiom is to append error messages to a log file as the build proceeds:

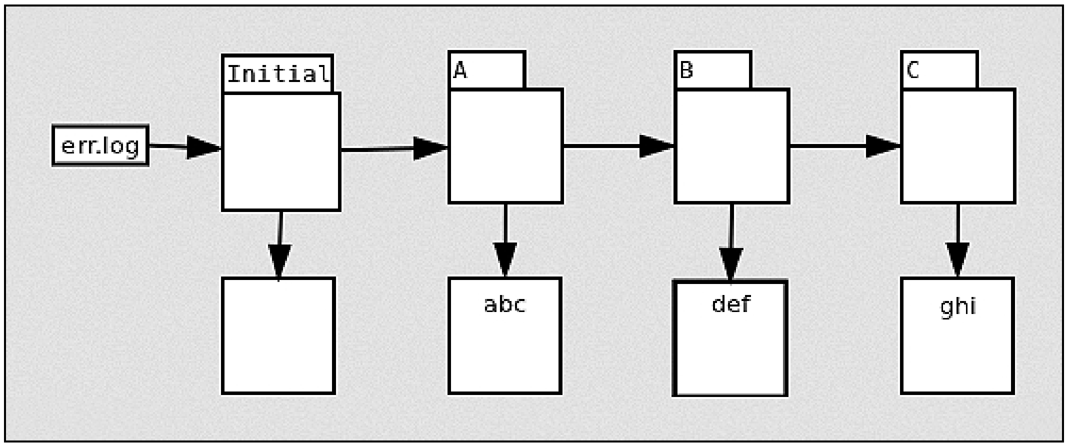
```
$(OUTDIR)/foo.o: foo.cpp
    @g++ -o $@ $^ 2>> err.log
```

Each append operation implicitly depends on the previous appends to the file—because the system needs to know which offset the new content should be written to if it does not know how big the file was to begin with. In terms of file versions, a naive implementation treating each append to the file as creating a complete new version of the file is possible:



Of course, the is that conflicts will occur if you try to run all of these jobs in parallel. Suppose that all three jobs, A, B and C start at the same time. They will each see the initial version, an empty file, but if run serially, only A would have seen that version. B would have seen the version created by A; C would have seen the version created by B.

This example is particularly interesting, because eMake cannot sort this out on its own: As long as the usage reported for `err.log` is the very generic "this file was modified, here's the new content" message normally used for changes to the content of an existing file, eMake has no choice but to declare conflicts and serialize these jobs. Fortunately, eMake is not limited to that simple usage record. The EFS can detect that each modification is strictly appending to the file (with no regard to the prior contents) and includes that detail in the usage report. Thus informed, eMake can record fragments of the file, rather than the entire file content:



Because eMake now knows that the jobs do not depend on the prior content of the file, it need not declare conflicts between the jobs, even if they run in parallel. As eMake commits the modifications from each job, it stitches the fragments together into a single file with each fragment in the correct order relative to the other pieces.

Directory-Read Conflicts

Directory-read operations are interesting from the perspective of conflict detection. Consider: What does it mean to read a directory? The directory has no content of its own, not in the way that a file does. Instead, the "content" of a directory is the list of files in that directory. To check for conflicts on a directory read, eMake must check whether the list of files that the reader job actually saw matches the list that it would have seen had it run in serial order—in essence, doing a simple conflict check on each of the files in the directory.

That is s conceptually easy to do, but the implications of doing so are significant: It means that eMake will declare a conflict on the directory read anytime any other job creates or deletes any file in that directory. Compare that to reads on ordinary files: You only get a conflict if the read happens before a write operation on the same file. With directories, you can get a conflict for modifications to other files entirely.

This is particularly problematic, because many tools actually perform directory reads in the background, and often those tools are not actually concerned with the complete directory contents. For example, a job that enumerates files matching `*.obj` in a directory is only interested in files ending with `.obj`. The creation of a file named `foo.a` in that directory should not affect the job at all.

Another problematic example comes from utilities that implement their own version of the getcwd() system call. If you want to create your own version, the algorithm looks something like this:

- Let `cwd` = ""
- Let `current` = "."
- Let `parent` = "./.."
- Stat `current` to get its inode number.
- Read `parent` until an entry matching that inode number is found.

- Add the name from that entry to `cwd`.
- Set `current` = `parent`.
- Set `parent` = `parent` + "/.."
- Repeat starting with step 4.

By following this algorithm, the program can construct an absolute path for the current working directory. The problem is that the program has a read operation on every directory between the current directory and the root of the file system. If eMake strictly adhered to conflict checking on directory reads, a job that used such a tool would be serialized against every job that created or deleted any file in any of those directories.

For this reason, eMake deliberately ignores conflicts on directory-read operations by default. Most of the time, this is safe to do, because often tools do not need a completely accurate list of the files in the directory. And even if the tool does require a perfectly correct list, the tool follows the directory read with reads of the files that it finds. This means that you can ensure correct behavior by running the build one time with a single agent to ensure the directory contents are correct when the job runs. That run will produce history based on the file reads, so subsequent builds can run with many agents and still produce correct results.

You can also do one of the following:

- Use `-emake-readdir-conflicts=1` to force eMake to honor directory-read conflicts for the build.
- Use the `#pragma readdirconflicts` pragma to enable directory-read conflicts on a per-job basis. You can apply it to targets or rules in your makefiles. This pragma has less overhead than –`-emake-readdir-conflicts=1` (which enables directory-read conflicts for an entire build). You can use this pragma in pragma addendum files as well as in standard makefiles.

Chapter 8: Annotation

As eMake runs a build, it discovers a large amount of information about the build structure. This information can be written to an *annotation* file for use after the build completes. Annotation is represented as an XML document to make parsing easy.

eMake collects many different types of information about the build depending on various eMake command-line options. This information includes:

- Makefile structure
- Commands and command output
- List of file accesses by each job
- Dependencies between jobs
- Detailed timing measurements
- eMake invocation arguments and environment
- Performance metrics

eMake can also be configured to upload annotation information to the Cluster Manager for centralized reporting.

Configuring eMake to Generate an Annotation File

By default, eMake collects configuration information and performance metrics only, which it sends to the Cluster Manager at the end of the build. This data is used to display reports on the Build Details page.

You can configure eMake to collect additional information. This information is written to an XML file in the build directory (`emake.xml` by default). The `--emake-annodetail` option controls the amount of information that eMake collects. Default annotation detail is determined by the build class for that build.

Following are the annotation detail arguments:

Argument	Description
`basic`	Information about every command run by the build. Detailed information about each "job" in the build is recorded, including command arguments, output, exit code, timing, and source location. Also, the build structure is a tree where each recursive make level is represented in the XML output. If the JobCache feature is enabled, basic annotation includes annotation about cache hits and misses. For details, see the "Interpreting Job Cache Annotation Information on page 6-13" section in the "Performance Optimization on page 6-1" chapter.
`env`	Environment variable modifications
`file`	Files read or written by each job

Argument	Description
`history`	Missing serializations discovered by eMake. This includes information about which file caused two jobs to become serialized by the eMake history mechanism
`lookup`	Files that were looked up by each job. *This mode can cause the annotation file to become quite large.*
`md5`	Computes MD5 checksums for files read and written by the build, and includes that information as an MD5 attribute on appropriate `<op>` tags. The operation types that will include the checksum are read, create, and modify. No checksum is generated or emitted for operations on directories, symlinks. or append operations. If a read file was appended to, and the read occurs before the appended update is committed, a zero checksum appears on that read operation (by design because reading files that were appended to occurs rarely). This argument implies "file" level annotation. This mode is configurable through the command line only; it is not available on the web interface.
`registry`	Registry operations
`waiting`	Complete dependency graph for the build

Any detail argument enables "basic" annotation automatically.

The `--emake-annoupload` option controls whether eMake sends a copy of the annotation file to the Cluster Manager as the build runs. By default, eMake sends minimal information to the Cluster Manager, even if more detailed annotation is enabled. eMake sends the full annotation file if annotation uploading is configured by the build class or via the eMake command line.

Note: You cannot disable `mergestreams` if you enable annotation. Enabling annotation automatically enables `mergestreams`, even if it was explicitly disabled on the command line.

Annotation File Splitting

Because of limitations in the 32-bit version of the ElectricInsight tool, eMake as well as Electrify automatically partition annotation files into 1.6 GB "chunks." The first chunk is named using the file name that you specify with the `--emake-annofile` option or with "emake.xml," if `--emake-annofile` is not specified. The second chunk uses that name as the base but adds the suffix _1, the third chunk adds the suffix _2, and so on. For example, a four-part annotation file might consist of files named emake.xml, emake.xml_1, emake.xml_2, and emake.xml_3.

No special action is required to load a multipart annotation file into ElectricInsight. If all parts are in the same directory, ElectricInsight automatically finds and loads the content of each file—simply specify the name of the first chunk when opening the file in ElectricInsight.

For loading large annotation files, Electric Cloud recommends the 64-bit version of ElectricInsight.

Working with Annotation Files

The simplest way to use an eMake annotation file is to load it into ElectricInsight. This tool lets you see a graphical representation of the build, search the annotation file for interesting patterns, and perform sophisticated build analysis using its built-in reporting tools.

Creating Tools for Tasks That Use Annotation Output

You can write your own tools to perform simple tasks that use annotation output. For example, reporting on failures in the build by looking for “failed” elements inside job elements and then reporting details about the failed job such as the commands, their output, and the line of the makefile that contains the rule for the command. See the DTD below or the annotation file format.

Annotation XML DTD

```
<!-- build.dtd -->
<!-- The DTD for Emake's annotated output. -->
<!-- -->
<!-- Copyright (c) 2004-2008 Electric Cloud, Inc. All rights reserved. -->

<!ENTITY % hexnum "CDATA">
<!ENTITY % job "(message*, job)">
<!ENTITY % valueName "name NMTOKEN #REQUIRED">
<!-- Can't use NMTOKEN because Windows has environment variables like
    "=D:". -->
<!ENTITY % envValueName "name CDATA #REQUIRED">

<!ELEMENT build
    (properties?, environment?, (message* | make)+, fs?, metrics? )
>
<!ATTLIST build
    id    CDATA #REQUIRED
    cm    CDATA #IMPLIED
    start CDATA #REQUIRED
>
<!-- Out of band build messages -->

<!ELEMENT message (#PCDATA) >
<!ATTLIST message
    thread    %hexnum;             #REQUIRED
    time      CDATA                #REQUIRED
    code      CDATA                #REQUIRED
    severity  ( warning | error ) #REQUIRED
>
<!-- Properties list -->

<!ELEMENT properties (property*) >
<!ELEMENT property   (#PCDATA) >
<!ATTLIST property
    %valueName;
>
<!-- Environment list -->

<!ELEMENT environment (var*) >
<!ELEMENT var   (#PCDATA) >
<!ATTLIST var
```

```
    %envValueName;
    op ( add | modify | delete ) "add"
>
<!-- File system dump -->

<!ELEMENT fs      (roots, symRoots, (content|name)*) >
<!ELEMENT roots   (root+) >
<!ELEMENT root    (#PCDATA) >
<!ATTLIST root
    nameid CDATA #REQUIRED
>
<!ELEMENT symRoots (symRoot*) >
<!ELEMENT symRoot (#PCDATA) >
<!ATTLIST symRoot
    symLinkPath CDATA #REQUIRED
>
<!ELEMENT content (contentver+)>
<!ATTLIST content
    contentid CDATA #REQUIRED
>
<!ELEMENT contentver EMPTY>
<!ATTLIST contentver
    job CDATA #REQUIRED
>
<!ELEMENT name (namever*) >
<!ATTLIST name
    nameid CDATA #REQUIRED
    dir    CDATA #REQUIRED
    name   CDATA #REQUIRED
>
<!ELEMENT namever EMPTY>
<!ATTLIST namever
    job       CDATA #REQUIRED
    contentid CDATA #REQUIRED
>
<!-- Metrics list -->

<!ELEMENT metrics (metric*) >
<!ELEMENT metric  (#PCDATA) >
<!ATTLIST metric
    %valueName;
>
<!-- Make subtree -->

<!ELEMENT make
    (environment?, ( message | job | make )*)
>
<!ATTLIST make
    level CDATA #REQUIRED
    cmd   CDATA #REQUIRED
    cwd   CDATA #REQUIRED
    mode  ( gmake | nmake | symbian ) #REQUIRED
>
<!-- Job -->

<!ELEMENT job
    (environment?,(output | command | conflict)*,depList?,opList?,
registryOpList?,timing+,failed?,waitingJobs?)
```

```
>
<!ATTLIST job
    thread    %hexnum;  #REQUIRED
    id        ID    #REQUIRED
    status    ( normal | rerun | conflict | reverted | skipped ) "normal"
    type      ( continuation | end | exist |
                follow | parse | remake | rule ) #REQUIRED
    name      CDATA #IMPLIED
    file      CDATA #IMPLIED
    line      CDATA #IMPLIED
    neededby  IDREF #IMPLIED
    partof    IDREF #IMPLIED
    node      CDATA #IMPLIED
>
<!-- Command and related output, output blocks can contain nested -->
<!-- make subtrees in local mode. -->

<!ELEMENT command
    (argv,inline*,(output | make)*)
>
<!ATTLIST command
    line  CDATA #IMPLIED
>
<!ELEMENT argv (#PCDATA)>
<!ELEMENT inline (#PCDATA)>
<!ATTLIST inline
    file CDATA #REQUIRED
>
`<!ELEMENT output (#PCDATA)>
<!ATTLIST output
    src   ( prog | make ) "make"
>
<!-- Conflict description -->

<!ELEMENT conflict EMPTY>
<!ATTLIST conflict
    type       ( file | cascade | name | key | value ) "cascade"
    writejob   IDREF #IMPLIED
    file       CDATA #IMPLIED
    rerunby    IDREF #IMPLIED
    hkey       CDATA #IMPLIED
    path       CDATA #IMPLIED
    value      CDATA #IMPLIED
>
<!-- Job failure code -->

<!ELEMENT failed EMPTY>
<!ATTLIST failed
    code CDATA #REQUIRED
>
<!-- List of jobs waiting for this job, local mode only -->

<!ELEMENT waitingJobs EMPTY>
<!ATTLIST waitingJobs
    idList IDREFS #IMPLIED
>
<!-- Start and stop times of this job -->
```

```
<!ELEMENT timing EMPTY>
<!ATTLIST timing
    invoked   CDATA #REQUIRED
    completed CDATA #REQUIRED
    node      CDATA #IMPLIED
>
<!-- Dependency list, only used when annoDetail includes 'history' -->

<!ELEMENT depList (dep*)>
<!ELEMENT dep     EMPTY>
<!ATTLIST dep
    writejob IDREF #REQUIRED
    file     CDATA #REQUIRED
>
<!-- Operation list, only present when annoDetail includes -->
<!-- 'file' or 'lookup' -->

<!ELEMENT opList (op*)>
<!ELEMENT op     EMPTY>
<!ATTLIST op
    type     ( lookup | read | create | modify | unlink | rename |
               link | modifyAttrs | append | blindcreate ) #REQUIRED
    file     CDATA #REQUIRED
    other    CDATA #IMPLIED
    found    ( 1 | 0 ) "1"
    isdir    ( 1 | 0 ) "0"
    filetype ( file | symlink | dir ) "file"
    atts     CDATA #IMPLIED
>
<!-- Registry operation list, only present when annoDetail includes -->
<!-- 'registry' -->

<!ELEMENT registryOpList (regop*)>
<!ELEMENT regop     (#PCDATA)>
<!ATTLIST regop
    type     ( createkey | deletekey | setvalue | deletevalue |
               lookupkey | readkey ) #REQUIRED
    hkey     CDATA #REQUIRED
    path     CDATA #REQUIRED
    name     CDATA #IMPLIED
    datatype ( none | sz | expandsz | binary | dword | dwordbe |
               link | multisz | resourcelist | resourcedesc |
               resourcereqs | qword ) "none"
>
```

Metrics in Annotation Files

The following values are available when you select the `Metrics` option from the drop-down menu on the **Build Details** page in the web interface. Some performance metrics are available only with `--emake-debug=g`.

Timer Annotation

Most timers are be available unless you use `--emake-debug=g` (for profiling). These timers correspond to the amount of time that eMake spent in certain areas of the code or in a certain state.

Metric	Description
timer:agentManager.should RequestAgents	Time spent checking to see if eMake should be requesting agents, which involves talking to the Cluster Manager
timer:agentManager.startup	Time spent for the Agent Manager to start up
timer:agentManager.stop	Time spent for the Agent Manager to shut down
timer:agentManager.work	Time spent with the Agent Manager actively doing work outside shouldRequestAgents
timer:annoUpload.startup	Time spent starting up the thread to upload annotation
timer:annoUpload.work	Time spent uploading annotation
timer:bench	Benchmark showing the cost of 100 invocations of the timer code (start/stop)
timer:directory.populate	Time spent making sure that eMake’s model of the directory contents is fully populated
timer:history.parsePrune	Time spent in parse jobs signaling stale history entries to prune stale events
timer:history.pruneFollowers	Time spent signaling stale submakes (for jobs that have followers) to be pruned
timer:history.pruneNo Follower	Time spent signaling stale submakes (for jobs that have no followers) to be pruned in job
timer:idle.agentManager	Time spent in the Agent Manager sleeping between shouldRequestAgents checks
timer:idle.agent ManagerRequest	Time spent in the Agent Manager waiting for the Cluster Manager to respond to a request for agents
timer:idle.agentRun	Time spent by worker threads waiting for requests from the agent
timer:idle.annoUpload	Time spent by the annotation upload thread waiting for data
timer:idle.noJobs	Time spent by worker threads waiting for a new runnable job to enter the job queue
timer:idle.untilCompleted	Time spent by the Terminator thread waiting for jobs to be completed
timer:idle.waitForAgent	Time spent by worker threads waiting for an agent to become available

Metric	Description
timer:jobCache.sharedMiss	(Shared JobCache only) Sum of seconds spent trying to find a match in the shared cache but ultimately failing to do so
timer:jobqueue	Time spent within the lock guarding the job queue
timer:Ledger.close	Time spent closing the Ledger and flushing data to disk
timer:Ledger.commit	Time spent committing Ledger data to the database
timer:Ledger.isUpToDate	Time spent querying the Ledger to find if a file is up to date
timer:Ledger.staleAttributes	Time spent by the Ledger code statting files to ensure recorded attributes match the actual attributes on disk
timer:Ledger.update	Time spent updating the Ledger
timer:main.commit	Time spent by the Terminator thread committing jobs, less time spent flushing deferred writes to disk
timer:main.history	Time spent reading and writing history files.
timer:main. lockedWriteToDisk	Time spent by the Terminator thread flushing deferred writes to disk
timer:main.terminate	Time spent by the Terminator thread terminating jobs
timer:main.writeToDisk	Time spent by the Terminator thread writing operations to disk, less the time covered by any of the other main.writeToDisk timers
timer:main.writeTodisk append	Time spent appending to existing files on disk
timer:main.writeTodisk createdir	Time spent creating directories on disk
timer:main.writeTodisk createdir.attrs	Time spent changing directory attributes on disk for new directories
timer:main.writeTodisk createdir.chown	Time spent changing file ownership on disk for new files
timer:main.writeTodisk createdir.chown	Time spent changing directory ownership on disk for new directories
timer:main.writeTodisk createdir.times	Time spent changing directory times on disk for new directories

Metric	Description
timer:main.writeTodisk createfile	Time spent writing file data to disk for new files
timer:main.writeTodisk createfile.attrs	Time spent changing file attributes on disk for new files
timer:main.writeTodisk createfile.times	Time spent changing file times on disk for new files
timer:main.writeTodisklink	Time spent creating links on disk
timer:main.writeTodisk modify	Time spent modifying existing files on disk
timer:main.writeTodisk modifyAttrs	Other time spent in writing attribute changes to disk (mostly notifying the file system that attributes have gone "stale")
timer:main.writeTodisk modifyAttrs.attrs	Time spent modifying attributes of existing files on disk
timer:main.writeTodisk modifyAttrs.chown	Time spent modifying ownership of existing files on disk
timer:main.writeTodisk modifyAttrs.times	Time spent modifying times of existing files on disk
timer:main.writeTodisk unlink	Time spent unlinking existing files on disk
timer:main.writeTodisk unlink.data	Time spent recording the fact that a file was removed
timer:main.writeTodisk unlink.tree	Time spent removing entire trees on disk
timer:mergeArchiveRefs	Time spent modifying word lists for multi-word archive references such as `lib(member1 member2 ...)`
timer:mutex.DirCache	Time spent waiting for a directory cache
timer:mutex.filedata.nodelist	Time spent within the lock guarding the list of agents allocated to the build
timer:mutex.jobcreate	Time spent within the lock used to synchronize the terminator and worker threads when creating jobs
timer:mutex.joblist	Time spent within the lock used to protect the job list

Metric	Description
timer:mutex.jobrunstate	Time spent within the lock used to coordinate starting and canceling jobs
timer:mutex.nodeinit	Time spent within the lock used to protect the list of hosts while initializing agents
timer:mutex.target	Time spent within the lock protecting failure tracking on a target
timer:node.putAllVersions. getShortName	Time spent getting file short names on Windows when doing an E2A_ PUT_ALL_VERSIONS
timer:node.setup	Time spent connecting to hosts and initializing them
timer:node.svc.getData	Time spent handling A2E_GET_FILE_DATA and A2E_RESEND_FILE_ DATA, not including the time spent sending data in response
timer:node.svc.getData. acquireLock	Time spent waiting for the ChainLock when handling A2E_GET_FILE_ DATA and A2E_RESEND_FILE_DATA
timer:node.svc.getData.copy	Time spent copying file data for E2A_LOAD_LOCAL_FILE. This is the local agent version of timer:node.svc.getData.send
timer:node.svc.getData. insideLock	Time spent holding the ChainLock when handling A2E_GET_FILE_ DATA and A2E_RESEND_FILE_DATA
timer:node.svc.getData.send	Time spent sending file data during E2A_PUT_FILE_DATA and E2A_ PUT_BIG_FILE_DATA.
timer:node.svc.getDir	Time spent sending directory entry data to the agent
timer:node.svc.getVersions	Time spent handling A2E_GET_ALL_VERSIONS requests
timer:node.svc.getVersions. acquireLock	Time spent waiting for the ChainLock when handling A2E_GET_ALL_ VERSIONS
timer:node.svc.getVersions. insideLock	Time spent holding the ChainLock when handling A2E_GET_ALL_ VERSIONS
timer:node.svc.runCommand	Time spent handling A2E_RUN_COMMAND requests
timer:usage.io	Time spent reading and responding to usage data
timer:usage.io.makedata	Time spent creating file data from usage
timer:usage.io. makedata.local	Time spent copying file data from usage reported by local agents; a subset of timer:usage.io.makedata
timer:usage.latency	Time spent dispatching incoming usage data and saving output files

Metric	Description
timer:usage.record	Time spent recording usage data, including resolving new name IDs, removing duplicate lookup records, and so on
timer:worker.continuationjob	Time spent by worker threads running continuationJobs
timer:worker.endjob	Time spent by worker threads running endJobs.
timer:worker.existencejob	Time spent by worker threads running existenceJobs
timer:worker.followjob	Time spent by worker threads running followJobs
timer:worker.invoke	Time spent invoking remote jobs
timer:worker.invokelocal	Time spent invoking local jobs
timer:worker.other	Otherwise unaccounted time spent by worker threads
timer:worker.parsejob	Time spent by worker threads running parseJobs.
timer:worker.remakejob	Time spent by worker threads running remakeJobs
timer:worker.rulejob	Time spent by worker threads running ruleJobs, less the time spent actually running commands and figuring out if it needs to run
timer:worker.rulejob. needtorun	Time spent by ruleJobs figuring out if they need to run
timer:worker. runcommands	Time spent by worker threads running commands in commandJobs
timer:worker. shouldRequestAgents	Time spent by worker threads checking to see if they should request agents
timer:worker.startup	Time spent by worker threads initializing
timer:worker.stop	Time spent shutting down worker threads
timerThreadCount	Number of threads in eMake

Other Annotation

Metric	Description
chainLatestReads	Number of times the latest version of an FSChain was requested
chainSerialReads	Number of times an FSChain was requested to match a particular spot in the serial order

Metric	Description
chainWrites	Number of new versions created, which occurs any time a new name is created, or the content of a file changes
clusterAvailability	Cluster availability percentage
compressBytesIn	Number of bytes passed in for compression
compressBytesOut	Number of bytes returned from compression
compressTime	Time spent compressing data
conflicts	Number of jobs that ran into conflicts and had to be rerun
decompressBytesIn	Number of bytes passed in for decompression
decompressBytesOut	Number of bytes returned from decompression
decompressTime	Time spent decompressing data
diskReadBytes	Number of bytes read from the local disk
diskReadTime	Time spent reading data from the local disk
diskReadWaitTime	Time spent waiting for data to be read from the local disk
diskWriteBytes	Number of bytes written to the local disk
diskWriteTime	Time spent writing data to the local disk
diskWriteWaitTime	Time spent waiting for data to be written to the local disk
duration	Duration of build (in seconds)
elapsed	The total elapsed time for the build
emptyNameVersions	Total number of Name versions with no associated Content. Available with `--emake-debug=g`
fitness	Indicates the "fitness" of the history, which is how closely the history used for a given build "matched" that build. The range of values is 0 (history did not match at all) to 1 (history already had information about all implicit dependencies) Available with `--emake-debug=g`
freezeTime	Time the job queue was frozen, which means only high priority items are taken of the queue

Metric	Description
hiddenWarningCount	Number of warning messages hidden by the eMake client and all remote parse jobs, with one count for every message number for which at least one message was hidden. The count does not include messages hidden by eMake stubs or rlocal-mode eMakes
jobcache.hit	Number of jobs for which eMake obtained job cache hits. A hit means that the job was replayed from the cache. This is the sum of the jobcache.hit.local and jobcache.hit.shared hits
jobcache.hit.local	(Shared JobCache only) Number of jobs for which eMake obtained job cache hits from the developer's local cache (instead of the shared cache). A hit means that the job was replayed from the local cache
jobcache.hit.shared	(Shared JobCache only) Number of jobs for which eMake obtained job cache hits from the shared cache (instead of the developer's local cache). A hit means that the job was replayed from the shared cache
jobcache.miss	Number of jobs for which a file system input changed since the last time the job was saved into the cache
jobcache.na	Number of jobs for which job caching was not applicable. For example, jobs for which the output files are not object files
jobcache.newslot	Number of jobs for which eMake created a new cache slot. This occurs when the previously-cached jobs differed from these jobs in their command-line arguments, relevant environment variables, or working directories. Together, these parameters of a job choose the slot for that job, which is where eMake stores the results of that job
jobcache.rootschanged	Number of jobs for which there is no natural mapping from the old eMake roots to the new eMake roots
jobcache.sharedmiss	(Shared JobCache only) Number times that eMake found a matching slot in the shared cache but determined it was a mismatch because a file system input changed
jobcache.sharednewslot	(Shared JobCache only) Number of times that eMake failed to find a matching slot in the shared cache
jobcache.uncacheable	Number of jobs for which eMake encountered an issue updating the job's relevant cache slot. Look for `ERROR` and `WARNING` messages in the console output from eMake. Jobs that failed, jobs for which caching was disabled by the `#pragma jobcache none` pragma, and jobs that did not access files inside the eMake root are included in this metric

Metric	Description
jobcache.unneeded	Number of jobs for which JobCache was enabled but not needed (because the target was already up to date according to ordinary GNU Make rules). The cache was not consulted, even though caching was requested for that target
jobcache.workloadsaved	Difference between agent usage (in seconds) with and without the JobCache feature. This represents an estimate of the agent usage saved by the JobCache feature
localAgentReadBytes	Bytes copied from local agents
localAgentReadTime	Time spent copying data from local agents
localAgentWriteBytes	Bytes copied to local agents
localAgentWriteTime	Time spent copying data to local agents
maxArenaCount	Peak active arenas during the build
maxMakeCount	Peak active make instances during the build
noAgentsWaitTime	Time spent waiting with no agents allocated to the build because the number of running agents has reached the license limit
noLicenseWaitTime	Time spent waiting for a build license because the number of concurrent builds has reached the license limit.
nodeReadBytes	Number of bytes read from agents
nodeReadTime	Time spent reading data from agents
nodeReadWaitTime	Time spent waiting for data to be read from agents
nodeWriteBytes	Number of bytes written to agents
nodeWriteTime	Time spent writing data to agents
nodeWriteWaitTime	Time spent waiting for data to be written to agents
OutputHistoryMD5	MD5 checksum of the history file that was generated by a build
runjobs	Number of jobs that did work
symlinkReads	Number of times the application read symlinks from the local disk
termDiskCopiedBytes	Number of bytes committed by copying. This should be a small number if possible. If you are using ClearCase and this number is nonzero, look into the `--emake-clearcase=vobs` option

Metric	Description
termDiskCopiedFiles	Number of files committed by copying. This number should be small if possible
termDiskMovedBytes	Number of bytes committed by moving. You want this to be big number if possible
termDiskMovedFiles	Number of files committed by moving. You want this to be a big number if possible
termDiskRemovedBytes	Number of bytes committed by remove (original file)
termDiskRemovedFiles	Number of files committed by remove (original file)
termDiskRemoved TmpBytes	Number of bytes committed by remove (temporary file)
termDiskRemoved TmpFiles	Number of files committed by remove (temporary file)
terminated	Number of jobs that ran to completion. These jobs might not necessarily have done anything
totalAgentCount	Total number of agents in cluster
totalChains	Total number of FSChains in the build—anything in the file system tracked by eMake for versioning is an FSChain, for example, file names, file contents. Because both names and contents are tracked, this should be at least twice the number of files accessed in the build
totalNameVersions	Total number of Name versions. Available with `--emake-debug=g`
totalVersions	Total number of FSChain versions. Available with `--emake-debug=g`
timer:jobcache.sharedMiss	(Shared JobCache only) Sum of seconds spent trying to find a match in the shared cache but ultimately failing to do so
usageBytes	Number of bytes received for usage data
workload	Sum of agent usage over all agents (in seconds). This value is also reported to the Cluster Manager
writeThrottleWaitTime	Time spent waiting for the write throttle (which is in place to avoid slowing the build by seeking the disk head back and forth too often)

Chapter 9: Third-Party Integrations

The following topics provide information about how ElectricAccelerator integrates with the following environments:

- ClearCase
- Coverity
- Cygwin
- Eclipse

Using ClearCase with ElectricAccelerator

If your build environment relies on the ClearCase source control system, there are some special considerations for running eMake. ClearCase views can be either "snapshot" or "dynamic."

- ClearCase *snapshot views* behave like a normal file system, so no special support is required.
- ClearCase *dynamic views* have non-standard file system behavior that requires explicit handling by eMake.

Note: ElectricAccelerator does not currently support ClearCase integration on Solaris x86. If you have need of this support, please contact your Electric Cloud sales representative. You can make use of ecclearcase_fake.so to provide information about your ClearCase setup through an `ini` file. Refer to eMake's "Fake" Interface for ClearCase on page 9-3.

Configuring ElectricAccelerator for ClearCase

ecclearcase Executable

Set the EMAKE_CLEARCASE_SERVER environment variable to the path of the ecclearcase executable. This ensures eMake can locate the correct version of ecclearcase if, for example, you are using 64-bit eMake.

LD_LIBRARY_PATH

Using ElectricAccelerator in a ClearCase environment requires your LD_LIBRARY_PATH (on UNIX) or PATH (on Windows) to contain a directory that includes libraries required to run "cleartool." Library file names for Windows or UNIX begin with "libatria" (Windows - *libatria.dll*, UNIX - *libatria.so).*

If you plan to use ClearCase with eMake, you must add the ClearCase shared libraries to the LD_LIBRARY_PATH on your system.

For sh:

```
LD_LIBRARY_PATH=/usr/atria/linux_x86/shlib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

For csh:

```
setenv LD_LIBRARY_PATH /usr/atria/linux_x86/shlib:${LD_LIBRARY_PATH}
```

(`/usr/atria/linux_x86/shlib` is an example and might differ on your system depending on what OS you use and where ClearCase is installed.)

To ensure ElectricAccelerator knows where ClearCase is installed, edit `/etc/ld.so.conf` to include the ClearCase installation location. As a second option, you can include the ClearCase installation location in LD_LIBRARY_PATH.

ClearCase Views on Agents

When ElectricAccelerator replicates a ClearCase view on an agent, it appears as a generic file system—ClearCase commands that run as part of a build will not work on the host, even if ClearCase is installed on that machine. The Electric File System masks ClearCase's VOB mounts. If your build runs ClearCase commands, these commands must be *runlocal* steps. For additional information, see Running a Local Job on the Make Machine on page 6-25.

When replicates a ClearCase view on an agent, it appears as a generic file system—ClearCase commands that run as part of a build will not work on the host, even if ClearCase is installed on that machine. The Electric File System masks ClearCase's VOB mounts. If your build runs ClearCase commands, these commands must be *runlocal* steps. For additional information, see Running a Local Job on the Make Machine on page 6-25.

Note: Because of the potential adverse interaction between two different file systems (ClearCase and ElectricAccelerator), Electric Cloud recommends that you do ***not*** install ClearCase on ElectricAccelerator Agent machines. If you must run ClearCase on an ElectricAccelerator Agent machine, ensure that whichever one you need to start and stop frequently is configured to start "second" at system startup time.

--emake-clearcase

The eMake command-line option `--emake-clearcase` controls which ClearCase features are supported for a build. By default, ClearCase integration for `rofs`, `symlink`, and `vobs` is disabled. To turn on support for these specific ClearCase features, if your build relies on these options, use `--emake-clearcase=LIST`, where `LIST` is a comma-separated list of one or more of the following values:

- `rofs` : detect read-only file systems

  eMake queries ClearCase for each file it accesses to determine whether the file should be considered 'read-only'.

- `symlink` : detect symbolic links (Windows only)

  eMake queries ClearCase for each file it accesses to determine whether the file is a ClearCase symbolic link.

- `vobs` : configure separate temporary directories for each vob

  Normally, eMake uses the `--emake-tmpdir` setting to determine where to place temporary directories for each device. With the 'vobs' option enabled, eMake automatically configures one directory per VOB. On Windows, eMake also communicates with ClearCase to determine which VOB a file belongs to so it can select the correct temporary directory.

Note: If `--emake-clearcase` is not specified on the command line and the environment variable `EMAKE_CLEARCASE` is present, eMake takes the options from the environment.

eMake's "Fake" Interface for ClearCase

In addition to a direct interface to ClearCase, eMake also provides a "fake" interface that allows the end user to pass information manually to eMake about the ClearCase environment. Normally, you invoke ClearCase functionality by specifying `--emake-clearcase=LIST` to eMake, at which point eMake attempts to load ecclearcase6.so and ecclearcase7.so (.dll on Windows). Whichever library successfully initializes in the ClearCase environment is used to talk to ClearCase through a provided API that is no longer maintained or supported. You can specify the precise library to load by setting the environment variable `EMAKE_CLEARCASE_LIBRARY` to the path to the desired library.

Under some conditions, the ClearCase API does not function properly. For this circumstance, eMake provides ecclearcase_fake.so (.dll on Windows). If you point EMAKE_CLEARCASE_LIBRARY to the fake interface, eMake loads that instead. The fake interface then loads the file specified in the environment by ECCLEARCASE_FAKE_INI, defaulting to ecclearcase_fake.ini. The `ini` file has two sections: `[vobs]` and `[attrs]`.

The `[vobs]` section maps a VOB path to a comma-separated set of attributes. Currently, `public` should be present for a public VOB and `ro` for read-only.

The `[attrs]` section maps a file name to *symlink***type*, where *symlink* might be empty if the file is not a symbolic link and *type* can be `null, version, directory_version, symbolic_link, view_private, view_derived, derived_object, checked_out_file, checked_out_dir`. If *symlink* is not empty, `symbolic_link` is assumed. If *type* is `version` or `directory_version` and `--emake-clearcase=rofs` is active, the EFS returns EROFS (or STATUS_ACCESS_DENIED on Windows) when an attempt is made to write the file.

If CLEARCASE_ROOT is set in the environment (as by `cleartool setview`), all `[attrs]` entries are tracked under their exact path as well as one with the CLEARCASE_ROOT prepended. If CLEARCASE_ROOT is set to `/view/testview`, setting `/vobs/test/symlink2` in `[attrs]` is the same as setting both `/vobs/test/symlink2` and `/view/testview/vobs/test/symlink2`.

Sample `ini` files for UNIX:

```
[vobs]
/vobs/test=public
/vobs/readonly=public,ro

[attrs]
/vobs/test/symlink2=symlink
/vobs/test/symlink/alpha=*directory_version
/vobs/test/symlink/beta=alpha
```

and Windows:

```
[vobs]
\test=public
\readonly=public,ro

[attrs]
S:/test/symlink2=symlink
S:/test/symlink/alpha=*directory_version
S:/test/symlink/beta=alpha
```

Where ClearCase Dynamic Views Affect eMake Behavior

Read-Only Mounts

ClearCase can mount files in a read-only mode, which means they appear to be writable, but any attempts to modify these files fail with a "read-only file system" (UNIX) or "access denied" (Windows) error message. Because eMake cannot tell whether a file is modifiable using normal file system interfaces, it does not know to disallow modifications performed by commands running on the agents. This activity leads to failures when eMake attempts to commit changes (incorrectly) allowed on the agent.

A simple test case:

```
unix% cleartool ls
Makefile
clock@@/main/2  Rule: /main/LATEST

unix% cat Makefile
all:
mv clock clock.old
```

The file "clock" is checked in to ClearCase. Makefile attempts to rename it. If you just run "make", it fails immediately, but can be instructed to ignore the error:

```
unix% make -i
mv clock clock.old
mv: cannot move `clock' to `clock.old': Read-only file system
make: [all] Error 1 (ignored)
```

Note that this file system is ***not*** mounted *read-only,* so a Makefile can be created. Because "clock" is checked-in, it cannot be renamed without checking it out first, and ROFS is the error ClearCase gives.

Now try this with eMake:

```
unix% emake --emake-root=/vobs -i
Starting build: 114626
mv clock clock.old
ERROR EC1124: Unable to rename file
/vobs/test/drivel/clock to /vobs/test/drivel/clock.old: Read-only
file system (error code 0x1e): Read-only file system
Interrupted build: 114626   Duration: 0:00 (m:s) Cluster
availability: 100%
```

Without activating ClearCase support, eMake does not know "clock" cannot be moved, so the operation succeeds on the agent, then fails when eMake attempts to commit it to disk. Specifying the "`-i`" flag to ignore errors will not work here.

```
unix% /home/user/Projects/4.2/i686_Linux/ecloud/emake
/emake --emake-clearcase=rofs --emake-root=/vobs -i
Starting build: 114630
mv clock clock.old
mv: cannot move `clock' to `clock.old': Read-only file system
make: [all] Error 1 (ignored)
Finished build: 114630   Duration: 0:00 (m:s)
Cluster availability: 100%
```

When eMake knows to replicate ClearCase's behavior, the error occurs on the host and can be handled normally.

Multiple VOBs

eMake writes uncommitted files into temporary directories, and moves them into their correct location after resolving any conflicts. eMake automatically places a temporary directory in the current working directory where it is invoked, and also creates a temporary directory in each location specified by the `--emake-tmpdir` option or the `EMAKE_TMPDIR` environment variable. When possible, eMake writes uncommitted files to the same physical device where the file will be saved when it is committed, which makes the commit operation a lightweight "rename" instead of a heavyweight "copy" operation.

Under ClearCase, each VOB functions as a separate physical disk, so to achieve optimal performance, a temporary directory must be specified for each VOB where the build writes files. `--emake-clearcase=vobs` sets up this directory for you automatically.

- On UNIX, each VOB has a distinct physical device ID, and this option is nothing more than a "shorthand" for specifying `EMAKE_TMPDIR=/vobs/foo:/vobs/bar:....` in the environment.
- On Windows, you must interface with ClearCase directly to make this distinction, so using `--emake-clearcase=vobs` is important to get the most speed for a build that writes to multiple VOBs.

Symbolic Links on Windows Platforms

On Windows, ClearCase conceals the nature of its symbolic links from other programs, so what is actually a single file appears to be two files to other programs. This situation hinders eMake's versioning mechanism as it tracks two separate chains of revisions for one underlying entity. A job's view of the file can get out of sync and cause build failures.

`--emake-clearcase=symlink` interfaces directly with ClearCase to determine whether a particular ClearCase file is a symbolic link and represents it on the agent as a reparse point, which is the native Windows equivalent of a symbolic link. All file operations are redirected to the target of the symbolic link to avoid synchronization problems. T

This issue does not occur on UNIX platforms, because ClearCase uses native file system support for symbolic links.

Following is a simple test case.

Beginning with a directory, "alpha", and a symlink to that directory, "beta":

```
windows% cleartool ls -d alpha beta
alpha@@/main/1   Rule: \main\LATEST
beta --> alpha
```

And a makefile:

```
all:
        @echo "Furshlugginer" > alpha/foo
        @echo "Potrzebie" > beta/foo
        @cat alpha/foo
```

```
windows% emake --emake-root=. -f symlink.mk
Starting build: 50070 Furshlugginer
Finished build: 50070   Duration: 0:02 (m:s)   Cluster availability: 100%

windows% emake --emake-root=. -f symlink.mk --emake-clearcase=symlink
Starting build: 50071 Potrzebie
Finished build: 50071   Duration: 0:01 (m:s)   Cluster availability: 100%
```

Explanation: ClearCase cannot tell the Windows file system that the symlink is a symlink, so `alpha/foo` and `beta/foo` appear to be distinct files. (On UNIX, this is not an issue, because symlinks are a standard operating system feature, which means that ClearCase can show them as such.) If a build does not contain ClearCase symbolic links, there is no reason to turn on the integration; if it does, eMake might assume that two different files exist when there is actually just one underlying file, in which case you must turn on the "symlink" part of the ElectricAccelerator ClearCase integration.

Performance Considerations

Running builds from ClearCase dynamic views can impose a considerable performance cost depending on the ClearCase configuration and your build. The best performance is achieved by using ClearCase snapshot views. If using snapshots is not possible, there are a few things to consider when setting up an eMake build.

Enabling the "symlink" or "rofs" options incurs a performance cost because of the need to communicate with the ClearCase server when accessing a file for the first time. Many builds do not need these features, even if they are running inside a ClearCase dynamic view, so consider leaving them turned off unless you encounter unexpected build failures.

Enabling the "vobs" option should have minimal performance cost, and might significantly speed up your build if build output is written back to your dynamic view.

Because of improved caching, eMake might provide a significant performance boost beyond that provided by running build steps in parallel. eMake caches much of the file system state, reducing the total number of requests to the ClearCase server during the build. Depending on how heavily loaded your ClearCase server is, this can significantly improve build performance. If you notice build speedups higher than you would expect given the number of agents in your cluster, improved caching might be the reason.

Using the "fake" interface for ClearCase (see eMake's "Fake" Interface for ClearCase on page 9-3 section), which lets you specify the details of VOBs and files in a static file, is much faster than communicating with ClearCase. This might suffice for many users.

Using Coverity with ElectricAccelerator

You can integrate Coverity Analysis for C/C++ into a makefile-based ElectricAccelerator build on Linux or Windows. For information about integrating Coverity with ElectricAccelerator and using Coverity with ElectricAccelerator, see the KB article *KBEA-00162 Integrating Coverity Analysis for C/C++ into an ElectricAccelerator Build*.

Using Cygwin with ElectricAccelerator (Windows Only)

Cygwin is a Linux-like environment for Windows that consists of two parts:

- A DLL (`cygwin1.dll`) that acts as a Linux API emulation layer, providing substantial Linux API functionality.
- A collection of tools that provide a Linux look and feel.

If your builds used gmake in a Cygwin environment, you might need to use eMake's `--emake-emulation=cygwin` option.

For more information about other Cygwin-specific eMake command-line options and corresponding environment variables, see Windows-Specific Commands on page 3-21. Specifically, the following command-line options:

```
--emake-cygwin=<Y|N|A>
```

and

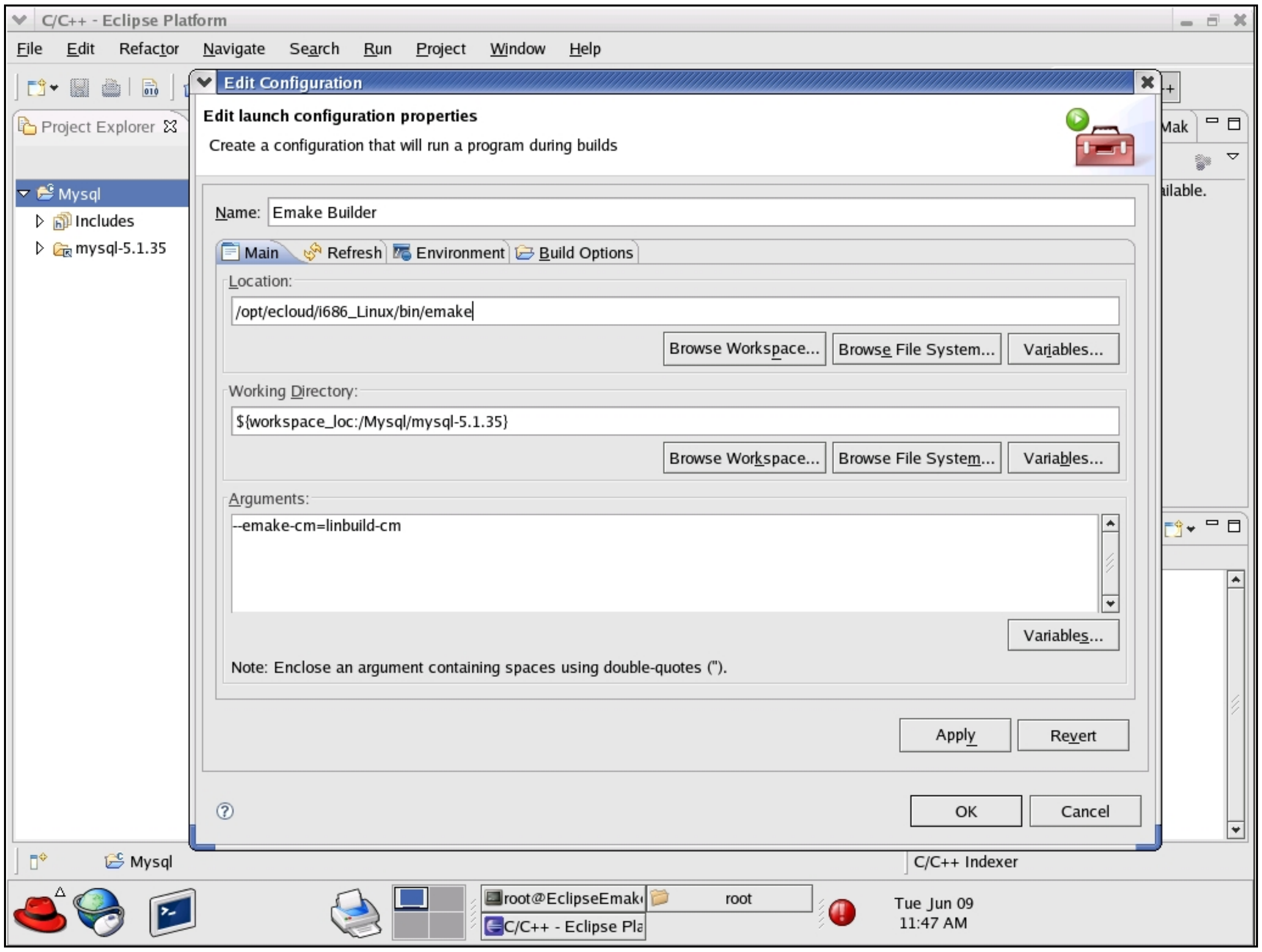
```
--emake-ignore-cygwin-mounts=<mounts>
```

Using Eclipse with ElectricAccelerator

To configure Eclipse to run eMake, follow this procedure:

1. Open your C++ project.
2. Go to the project's Properties > Builders and click **New**.
3. Select **Program** and click **OK**.
4. Fill in the following information for the new builder under the Main tab:
   - Name
   - Location (the full path to `emake`, which is OS dependent)
   - Working Directory
   - Arguments (arguments are specific to your configuration)

   The following screenshot illustrates the Edit Configuration dialog.


Edit Configuration

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: Emake Builder

Location:
/opt/ecloud/i686_Linux/bin/emake

Working Directory:
${workspace_loc:/Mysql/mysql-5.1.35}

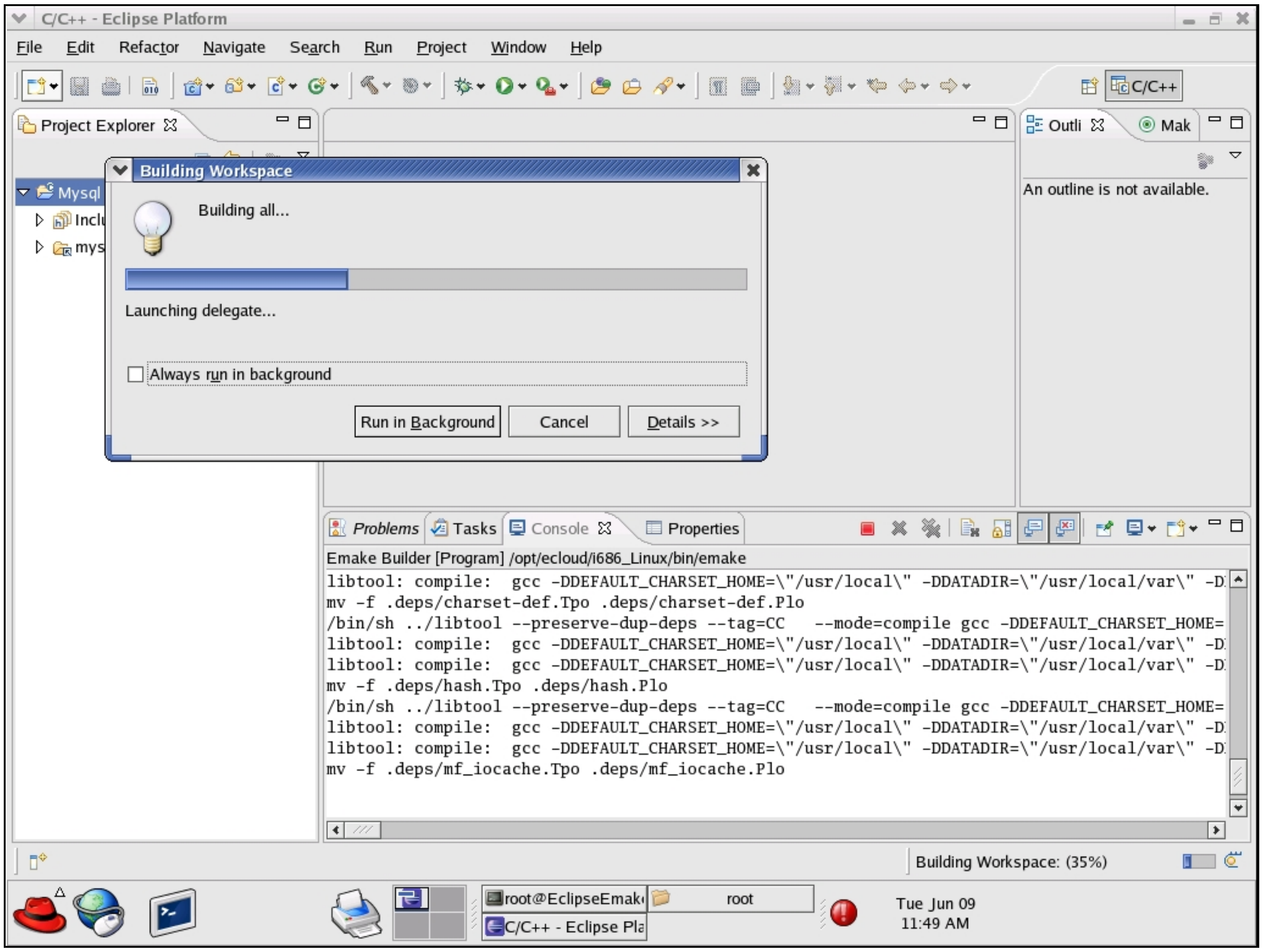
Arguments:
--emake-cm=linbuild-cm

Note: Enclose an argument containing spaces using double-quotes (").

5. Click the Build Options tab. Enable Run the builder for the following *only*:
   - After a “Clean”
   - During manual builds
   - During auto builds
6. Click **OK**. Your new builder is displayed in the Builders pane.
7. Create another builder for “cleans” only. On its Main tab, ensure `clean` is included for Arguments. On its Build Options tab, enable Run the builder for the following *only*:
   - During a “Clean”
8. Click **OK**. Your second builder is displayed in the Builders pane.
9. Deselect CDT Builder in the Builders pane and then click **OK**.

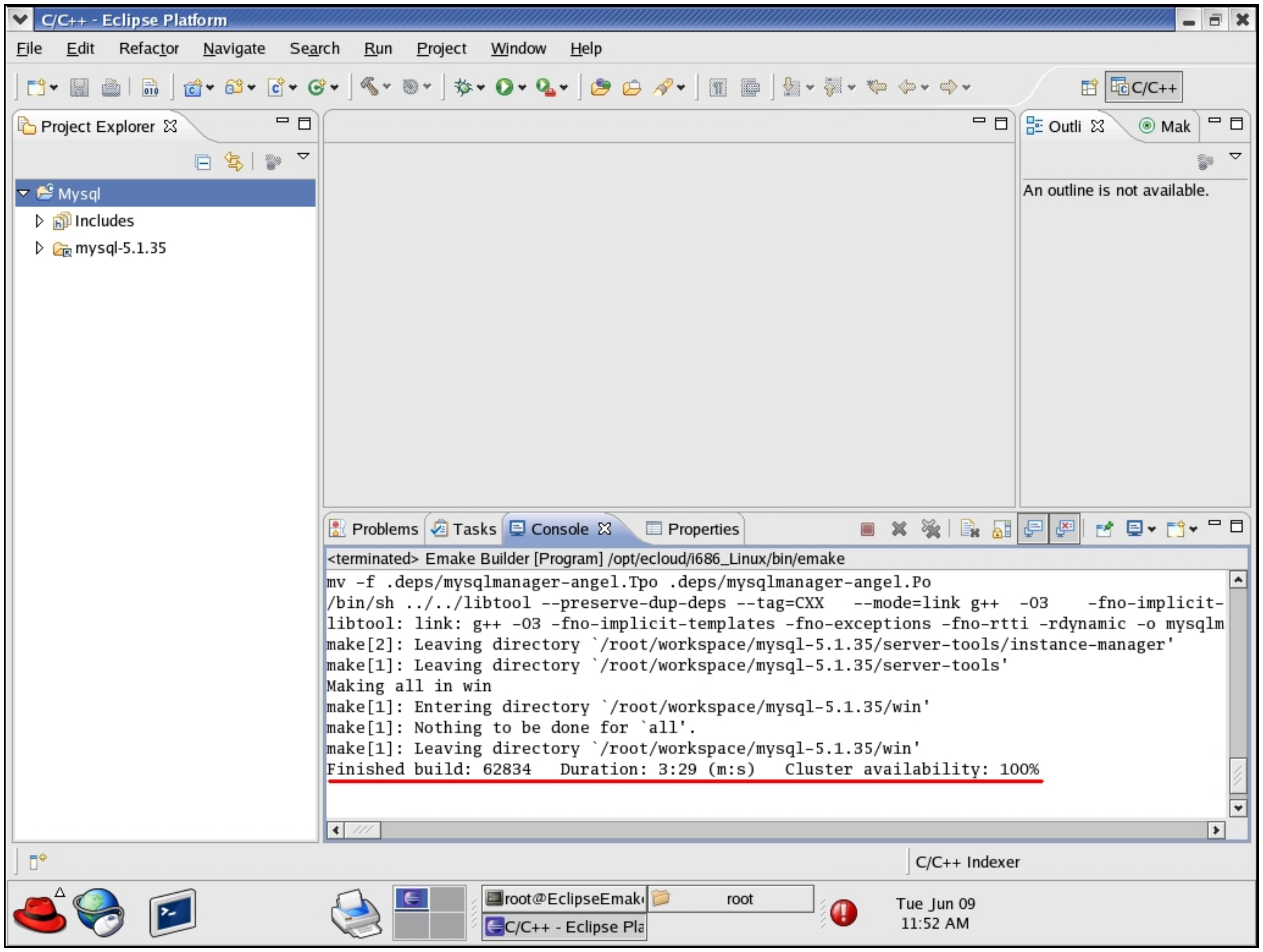
Now you can build your project. Click **Project > Build all**.

The following screenshot illustrates a build in progress.



```
Emake Builder [Program] /opt/ecloud/i686_Linux/bin/emake
libtool: compile:  gcc -DDEFAULT_CHARSET_HOME=\"/usr/local\" -DDATADIR=\"/usr/local/var\" -D
mv -f .deps/charset-def.Tpo .deps/charset-def.Plo
/bin/sh ../libtool --preserve-dup-deps --tag=CC   --mode=compile gcc -DDEFAULT_CHARSET_HOME=
libtool: compile:  gcc -DDEFAULT_CHARSET_HOME=\"/usr/local\" -DDATADIR=\"/usr/local/var\" -D
libtool: compile:  gcc -DDEFAULT_CHARSET_HOME=\"/usr/local\" -DDATADIR=\"/usr/local/var\" -D
mv -f .deps/hash.Tpo .deps/hash.Plo
/bin/sh ../libtool --preserve-dup-deps --tag=CC   --mode=compile gcc -DDEFAULT_CHARSET_HOME=
libtool: compile:  gcc -DDEFAULT_CHARSET_HOME=\"/usr/local\" -DDATADIR=\"/usr/local/var\" -D
libtool: compile:  gcc -DDEFAULT_CHARSET_HOME=\"/usr/local\" -DDATADIR=\"/usr/local/var\" -D
mv -f .deps/mf_iocache.Tpo .deps/mf_iocache.Plo
```

The following screenshot illustrates a successfully completed build.



An outline is not available.

```
<terminated> Emake Builder [Program] /opt/ecloud/i686_Linux/bin/emake
mv -f .deps/mysqlmanager-angel.Tpo .deps/mysqlmanager-angel.Po
/bin/sh ../../libtool --preserve-dup-deps --tag=CXX   --mode=link g++  -O3    -fno-implicit-
libtool: link: g++ -O3 -fno-implicit-templates -fno-exceptions -fno-rtti -rdynamic -o mysqlm
make[2]: Leaving directory `/root/workspace/mysql-5.1.35/server-tools/instance-manager'
make[1]: Leaving directory `/root/workspace/mysql-5.1.35/server-tools'
Making all in win
make[1]: Entering directory `/root/workspace/mysql-5.1.35/win'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/root/workspace/mysql-5.1.35/win'
Finished build: 62834   Duration: 3:29 (m:s)   Cluster availability: 100%
```

Chapter 10: Electrify

Electrify accelerates builds by parallelizing the build process and distributing build steps across clustered resources.

Limitations

- The tool that you want to monitor must provide parallel support, such as SCons.
- Electrify does not provide any of eMake's dependency detection or correction features. The build tools you use with Electrify must be capable of accurate parallel execution on their own.
- The information written into annotation is more limited with Electrify than what is provided by eMake. Electrify annotation provides information only on the commands executed on the cluster, including command lines, file usage, and raw command output. Electrify does not provide information about dependencies, job relationships, targets, or other logical build structure data.

Recommendations

Electric Cloud has evaluated the following build tools for use with Electrify.

- SCons—Electric Cloud recommends using SCons with Electrify. Using SCons does not have any known limitations.
- Ant—Electric Cloud does not currently recommend using Ant with Electrify.

Electrify as Part of the Build Process

The following sections describe how to run builds using Electrify.

Running Electrify on Windows

Example

```
electrify [args] other tools' command line
```

Running Electrify on Linux

Example

```
electrify [arguments] other tools' command line
```

SCons Example

```
electrify --emake-cm=<CM name> --electrify-remote=g++:gcc:ranlib:ar scons -j 4
```

GNU Make C++ Example

```
electrify --emake-cm=<CM name> --electrify-remote=g++:gcc:<any other tools used
in the build> make -j 4 -f makefile
```

Important Reminders About Electrify

- Ensure `cl.exe`, `link.exe`, and so on, are those of Microsoft Visual Studio. The wrapper application might have changed them to its version.
- Though all of the usual eMake arguments are available, Electrify uses only a subset of them.
- On 64-bit Windows platforms, if you did not install ElectricAccelerator in its default install location, you must specify the complete location of `electrifymon.exe` (including the executable name) for the `EMAKE_ELECTRIFYMON` environment variable.
  For example, if your custom install location is `C:\programs\ECloud`, then set the `EMAKE_ELECTRIFYMON` environment variable using:
  `set EMAKE_ELECTRIFYMON = C:\programs\ECloud \i686_win32\64\bin\electrifymon.exe`.
- On UNIX platforms, electrifymon must locate `electrifymon.so` so it can tell monitored programs to load the monitoring library that reports back to electrifymon. By default, electrifymon looks in the following locations:

Platform	32-bit	64-bit
Linux	/opt/ecloud/i686_Linux/32/lib	/opt/ecloud/i686_Linux/64/lib
Solaris (SPARC)	/opt/ecloud/sun4u_SunOS/lib	/opt/ecloud/sun4u_SunOS/64/lib
Solaris (x86)	/opt/ecloud/i686_SunOS.5.10/lib	/opt/ecloud/i686_SunOS.5.10/64/lib

  You can override these locations using the following environment variables: `ELECTRIFYMON32DIR` and `ELECTRIFYMON64DIR`.
- On Linux, you can change the mode that Electrify uses for monitoring. Use `--emake-electrify=<`*`mode`*`>`, where mode can be: `preload` for LD_PRELOAD intercept, or `trace` for ptrace.

Electrify Arguments

All arguments are optional.

`--electrify-remote=<`*`x;y`*`>`	x and y are commands that are distributed to the cluster. Use the command's full name, such as `cl.exe`, `link.exe`, `gcc.exe`, without the path. The name is case insensitive. In a Cygwin environment, you can use ':' (colon) instead of ';' (semicolon). Limited to 2048 characters Environment variable: `ELECTRIFY_REMOTE`
`--electrify-not-remote=<`*`x;y;`*`>`	x and y are commands that are not distributed to the cluster. Use the command's full name, such as `cl.exe`, `link.exe`, `gcc.exe`, without the path. The name is case insensitive. In a Cygwin environment, you can use ':' (colon) instead of ';' (semicolon). `--electrify-not-remote` and `--electrify-remote` are mutually exclusive. If you use `--electrify-not-remote`, all other tools' command lines are executed remotely, by default. Generally, this is not what you want, so to do this, you must add a command to this list.
`--electrify-not-intercept=<`*`x;y;`*`>`	xxx and yyy are commands that you do not want to be monitored, meaning the monitor process does not inject a dll to them and their child processes, so they will *not* be distributed. Environment variable: `ELECTRIFY_NOT_INTERCEPT`
`--electrify-log=<`*`fullpath`*`>`	fullpath is the path of the file you want to log. This logs all process creation and interception information. Environment variable: `ELECTRIFY_LOG`

`--electrify-localfile=<x>`	(Windows only) Integrates local file access (create, rename, and so on) by locally running tools with the remote file system. You can set x to two different flags: `NT` or `y`. Set `nt` if you want to monitor undocumented low-level file access Nt functions. This monitors the following functions: NtCreateFile, NtDeleteFile, NtClose, NtWriteFile, and NtSetInformationFile. Though this includes only five functions, their functionality is rich, so this selection includes nearly all scenarios where the local file system changes. Set `y` to monitor documented Win32 APIs for file access. This monitors the following Win32 APIs: CreateFileW, CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryW, DeleteFileA, DeleteFileW, MoveFileA, MoveFileExA, MoveFileExW, MoveFileW, RemoveDirectoryA, RemoveDirectoryW, SetFileAttributesA, and SetFileAttributesW. Though there are many functions, their functionality is less than Nt functions, particularly because some tools such as Cygwin cp.exe use NtCreateFile and so on. In general, use the `y` flag for testing purposes only.

`--electrify-allow-regexp=`
`<perl-regular-expression>`
	(Linux only) Sends all processes to the cluster whose full command-line matches the regular expression.

`--electrify-deny-regexp=`
`<perl-regular-expression>`
	(Linux only) The processes whose command-line match the expression will be executed locally. You can use this after the `--electrify-allow-regexp` option to tune the selection of processes that are sent to the cluster more precisely.

Using Whole Command-Line Matching and efpredict

Whole Command-Line Matching

On Linux, you can use a process's entire command-line to determine if it should be sent to the cluster for execution. Prior to Accelerator v7.0, the only way to discriminate was by the process name. Unfortunately, this did not work well for scripting languages or for languages that run in a VM, such as Java, because the process name is always 'java' no matter which particular program is being executed. This means that in previous versions you could send all Java programs to the cluster or none, and that was the limit of your discretion.

Additional Electrify command-line options:

```
--electrify-allow-regexp=<perl-regular-expression>
```

```
--electrify-deny-regexp=<perl-regular-expression>
```

These options allow you to specify which sub-processes to execute in the cluster. You use a Perl-style regular expression that is matched against both the process name and all of its arguments, such as the name of the script or JAR file that is being executed. When Electrify detects that a process is started, it constructs the command line for that process by joining all of the components of its "argv" array together with spaces and then applying the list of "allow" and "deny" regular expressions in the sequence that they were supplied on the command line.

Whole Command-Line Matching Example:

You have three processes:

```
java -jar runstep.jar -x86
```

```
java -jar otherjar.jar
```

```
java -jar runstep.jar -armv7
```

The following options send the first process to the cluster but not the second or third:

```
--electrify-regexp-allow="[^ ]+java\s.*runstep.jar.*"  --electrify-regexp-
deny=".*\-armv7.*"
```

Prior to Accelerator v7.0, you were only able to send all or none.

Initially, a process is considered to be for local execution only, but successive 'allow-regexp' options can change this state if any of them match. Any "deny-regexp" in the sequence will, if it matches, short-circuit the decision immediately and cause that process to be executed locally.

efpredict

efpredict helps you verify that the expressions you entered actually select the correct processes. Without efpredict, you would need to perform a full build and then examine the annotation file to see if the correct decisions were made. You would have to repeat this process each time there were any mistakes, and it could result in a long process. Instead, you can test settings with efpredict. Provide the same options as you would for Electrify and then enter command-lines into efpredict's standard input to see if it selects them for local or cluster execution. One easy way to do this is to pipe an old build log into efpredict. Check the resulting output visually to see if the desired processes were executed remotely.

efpredict Example:

```
cat oldlog | efpredict --electrify-regexp-allow="[^ ]+java\s.*runstep.jar.*"  --
electrify-regexp-deny=".*\-armv7.*"
```

gives this output:

```
remote_allow: java -jar runstep.jar -x86

remote_deny: java -jar otherjar.jar

remote_deny: java -jar runstep.jar -armv7
```

Important Notes

- Whole command-line matching is Linux only
- efpredict is Linux only
- If a process was executed by a shell, variables will be expanded, quotes will be removed, and white-space between tokens will be replaced with single spaces before Electrify matches the process. This means that if you look at a process invocation in a shell script or makefile, that might not be the exact text that Electrify sees when it attempts to intercept the invocation of that process.

  For example, in a script you might see:

  ```
  'gcc        "$SOURCE/myfile.c"       -o "$OUTPUT/myfile.o" -c    '
  ```

  but when Electrify intercepts this and tries to reconstruct the command line, it will see:

  ```
  "gcc src/myfile.c -o out/myfile.o -c".
  ```

  Regular expressions must be written to match what Electrify will be able to see.
- One regexp can match many commands. For example, to send both the gcc and ld commands to the cluster you can use:

  ```
  --electrify-regexp-allow='[^ ]*((gcc)|(ld))(\s.*)?'
  ```

Additional Electrify Information

Selecting Commands to Parallelize

A large portion of the build process acceleration will be achieved through parallelizing a small number of specific commands, such as compiling and linking. Expending additional effort to select and parallelize many different additional commands might not result in a significant amount of further acceleration.

Note: If a file is created or modified by one or more parallelized commands, then you should parallelize all commands that use that file.

Using Electrify with GNU Make

If you intend to use Electrify with GNU Make, Electric Cloud recommends using eMake instead. eMake provides superior performance and correctness and full annotation information.

How an Electrify Build Differs from an eMake Build

An important difference between an eMake build and an Electrify build is what portion of the build activity occurs remotely versus locally. In an eMake build, effectively all build activity (except "#pragma runlocal" jobs) takes place on the cluster, where the EFS is used to monitor file system accesses and propagate changes made by one job to other jobs in the build.

In an Electrify build, a greater portion of the build activity takes place on the local system—at the very least, the build process itself (such as SCons) runs locally. Typically, file system modifications made by processes running locally are "invisible" to Electrify, and therefore to processes running on the cluster—just as "#pragma runlocal" jobs might make changes that are invisible to eMake. Electrifymon provides a means to update the virtual file system state in Electrify in response to file system modifications made by local processes.

If you know that a build will not run any processes locally that modify the file system, you need not use electrifymon when invoking Electrify. However, some build tools will themselves make changes to the file system (for example, when SCons employs its build-avoidance mechanism by copying a previously built object in lieu of invoking the compiler), so the safest choice is to use electrifymon to start.

In addition to file system monitoring, electrifymon on Windows provides a sophisticated mechanism for intercepting processes invocations and determining which processes to distribute to the cluster. On Linux, process interception is handled by the explicit use of proxy commands.

Chapter 11: Troubleshooting

The following topics discuss information to assist you with troubleshooting.

Topics:

- Agent Issues
- eMake Debug Log Levels
- Using the Annotation File to Troubleshoot Builds

Agent Errors Establishing the Virtual File System

Agent errors regarding the establishment of the virtual file system for a particular build will be displayed if there are at least three errors. These errors would occur during the initial setup of the agent's build-specific environment but before any particular build step is run on that agent. The most common type of error involves eMake roots or Cygwin mounts, where the virtual file system setup is specific to the build but not to any particular build step.

Agents Do Not Recognize Changes on Agent Machines

If you manually mount a file system or change automounter settings on agent machines *after* they are started, you must restart the Agents for them to recognize your changes.

eMake Debug Log Levels

This section discusses eMake debug log levels. Content was adapted from the "Electric Make debug log levels" blog post on http://blog.melski.net/ and was the most recent information available when the article was posted.

Disclaimer: eMake debug logs are intended for use by Electric Cloud engineering and support staff. Debug logging contents and availability are subject to change in any release, for any or no reason.

Often when analyzing builds executed with eMake, all of the information you need is in the annotation file—an easily digested XML file containing data such as the relationships between the jobs, the commands run, and the timing of each job. But sometimes you need more detail, and that is where the eMake debug log is useful.

Enabling eMake Debug Logging

To enable eMake debug logging, specify this pair of command-line arguments:

`--emake-debug=<`*`value`*`>` specifies the types of debug logging to enable. Provide a set of single-letter values, such as "`jng`".

`--emake-logfile=<`*`path`*`>` specifies the location of the debug log.

eMake Debug Log Level Descriptions

Available log levels:

a: agent allocation	l: ledger
c: cache	m: memory
e: environment	n: node
f: file system	o: parse output
g: profiling	p: parse
h: history	P: parse avoidance

j: job	r: parse relocation
L: nmake lexer	s: subbuild
	Y: security

a: agent allocation

Agent allocation logging provides detailed information about eMake's attempts to procure agents from the Cluster Manager during the build. If you think eMake might be stalled trying to acquire agents, allocation logging will help to understand what is happening.

c: cache

Cache logging records details about the file system cache used by eMake to accelerate parse jobs in cluster builds. For example, it logs when a directory's contents are added to the cache, and the result of lookups in the cache. Because it is only used during remote parse jobs, you must use it with the `--emake-rdebug=value` option. Use cache logging if you suspect a problem with the *cached local file system*.

e: environment

Environment logging augments node logging with a dump of the entire environment block for every job as it is sent to an agent. Normally this is omitted because it is quite verbose (could be as much as 32 KB per job). Generally, it is better to use env-level annotation, which is more compact and easier to parse.

f: file system

File system logging records numerous details about eMake's interaction with its versioned file system data structure. In particular, it logs every time that eMake looks up a file (when doing up-to-date checks, for example), and it logs every update to the versioned file system caused by file usage during the build's execution. This level of logging is very verbose, so it is not usually enabled. It is most often used when diagnosing issues related to the versioned file system and conflicts.

g: profiling

Profiling debug logging is one of the easiest-to-interpret and most useful types of debug logging. When enabled, eMake emits hundreds of performance metrics at the end of the build. This is a very lightweight logging level and is safe (and advisable) to enable for all builds.

h: history

History logging prints messages related to the data tracked in the eMake history file—both file system dependencies and autodep information. When history logging is enabled, eMake will print a message every time a dependency is added to the history file, and it will print information about the files checked during up-to-date checks based on autodep data. Enable history logging if you suspect a problem with autodep behavior.

j: job

Job logging prints minimal messages related to the creation and execution of jobs. For each job you will see a message when it starts running, when it finishes running, and when eMake checks the job for conflicts. If there is a conflict in the job, you will see a message about that, too. If you just want a general overview of how the build is progressing, j-level logging is a good choice.

L: nmake lexer

eMake uses a generated parser to process portions of NMAKE makefiles. Lexer debug logging enables the debug logging in that generated code. This is generally not useful to end users because it is too low-level.

l: ledger

Ledger debug logging prints information about build decisions based on data in the ledger file, as well as updates made to the ledger file. Enable it if you believe the ledger is not functioning correctly.

m: memory

When memory logging is enabled, eMake prints memory usage metrics to the debug log once per second. This includes the total process memory usage as well as current and peak memory usage grouped into several "buckets" that correspond to various types of data in eMake. For example, the "Operation" bucket indicates the amount of memory used to store file operations; the "Variable" bucket is the amount of memory used for makefile variables. This is most useful when you are experiencing an out-of-memory failure in eMake because it can provide guidance about how memory is being utilized during the build, and how quickly it is growing.

n:node

Node logging prints detailed information about all messages between eMake and the agents, including file system data and commands executed. Together with job logging, this can give a very comprehensive picture of the behavior of a build. However, node logging is extremely verbose, so enable it only when you are chasing a specific problem.

o: parse output

Parse output logging instructs eMake to preserve the raw result of parsing a makefile. The result is a binary file containing information about all targets, rules, dependencies, and variables extracted from makefiles read during a parse job. This can be useful when investigating parser incompatibility issues and scheduling issues (for example, if a rule is not being scheduled for execution when you expect). Note that this debug level only makes sense when parsing, which means you must specify it in the `--emake-rdebug` option. The parse results will be saved in the `--emake-rlogdir` directory, named as `parse_jobid.out`. Note that the directory might be on the local disk of the remote nodes, depending on the value you specify.

p: parse

Parse debug logging prints extremely detailed information about the reading and interpretation of makefiles during a parse job. This is most useful when investigating parser compatibility issues. This output is very verbose, so enable it only when you are pursuing a specific problem. Like parse output logging, this debug level only makes sense during parsing, which means you must specify it in the `--emake-rdebug` option. The parse log files will be saved in the `--emake-rlogdir` directory, named as `parse_jobid.dlog`. Note that the directory might be on the local disk of the remote nodes, depending on the value you specify.

P: parse avoidance

Parse avoidance logging indicates when a new parse is required, and if so, why it was required.

r: parse relocation

Parse relocation logging prints low-level information about the process of transmitting parse result data to eMake at the end of a parse job. It is used only internally when we the parse result format is being extended, so is unlikely to be of interest to end-users.

s: subbuild

Subbuild logging prints details about decisions made while using the eMake subbuild feature. Enable it if you believe that the subbuild feature is not working correctly.

Y: authentication

Authentication logging is a subset of node logging that prints only those messages related to authenticating eMake to agents and vice-versa. Enable this debug level, if you are having problems using the authentication feature.

Using the Annotation File to Troubleshoot Builds

Annotation helps you debug build problems by identifying performance issues and determining the reasons for rebuilding a “no-touch” build.

Determining Why a Target Was Rebuilt

The `<job>` tag includes a `<why>` tag with a reason as well as a `prereq` attribute if applicable. The `<why>` tag reports why a job ran, which lets you differentiate between eMake-only reasons (for example, reasons related to eDepend or Ledger) and make-related reasons (for example, the job target is absent or out of data).

Following is an example where the target did not exist before the job ran:

```
<job ...>
...
  <why reason="target-absent"/>
...
</job>
```

During an incremental build, a job probably ran because a dependency was newer than the target. The `<why>` tag includes a `prereq` attribute. For example:

```
<job ...>
...
  <why reason="prereq-newer" prereq="/path/to/hello.c"/>
...
</job>
```

The following example demonstrates the eDepend (`--emake-autodepend=1`) feature. The job was run because of this dependency:

```
<job ...>
...
  <why reason="autodep-prereq" prereq="/path/to/conf-obj-dir"/>
...
</job>
```

Following are the possible values for the `reason` attribute. These indicate the reasons that a job can be run:

```
always-make
autodep-prereq
double-colon-no-prereq
dryrun-prereq
intermediate-newer
invalid-charset
ledger
not-run
phony
prereq-absent
prereq-did-work
prereq-newer
target-absent
```

`double-colon-no-prereq` is gmake-only. `invalid-charset` and `prereq-did-work` are NMAKE-only. `autodep-prereq`, `intermediate-newer`, `prereq-absent`, `prereq-did-work`, and `prereq-newer` can have a `prereq` attribute. For details about the complete list of possible values, see the *ElectricAccelerator Annotation Guide* at http://docs.electric-cloud.com/accelerator_doc/AcceleratorIndex.html.

Profiling Metrics

Annotation includes profiling metrics. These are the same metrics that are in the debug log file when the `--emake-debug=g` option is set. (The metrics appear in annotation whether or not `--emake-debug=g` is used.) These metrics are intended for use by Electric Cloud technical support and engineering staff.

Profiling metrics are within the `<profile>` tag and appear exactly as they do in the debug log file.

Per-Job Performance Metrics

Annotation includes per-job performance metrics. These metrics track a variety of details about the performance of each job in the build. For example, metrics with the `A2E_` prefix reflect low-level details of the eMake/agent protocol, such as file data and metadata requests made by the agent on behalf of the commands run during the job. Not all metrics are for use by end users, but Electric Cloud technical support and engineering staff might use them for certain performance analyses.

Per-job performance metrics are within the `<jobMetrics>` tag as in the following example.

```
<job... >
...
   <jobMetrics>
      A2E_GET_ALL_VERSIONS 11
      A2E_GET_FILE_DATA 2
   </jobMetrics>
...
</job>
```

Index

#

#pragma

allserial 6-26

cache javadoc 6-16, 6-23

multi 4-12

noautodep 7-5

runlocal 6-25

A

Accelerator

component interactions 1-2

agent

errors 11-2

Android 6-2

annotation 8-1

metrics 8-6

timers 8-6

B

build

configuration 2-4

defining 2-2

environment 2-2

performance optimization 6-2

sample 3-24

sources 2-2

stopping 4-16

tools 2-2

with Electrify 10-1

build class 4-2

examples 4-3

build parts 1-5

building multiple classes simultaneously 4-12

C

ccache 3-6

ClearCase 9-2

Cluster Manager

host 3-6

command line options 3-8

commands that read from console 5-3

configuration

ccache 3-6

ClearCase 9-2

Cluster Manager 3-6

eMake temporary directory 6-28, 6-28

environment variables 3-5

tools 3-4

Coverity 9-7

Cygwin 9-8

D

delayed existence checks 5-13

dependency optimization 6-2

E

Eclipse 9-8

eDepend 7-2

- benefits 7-3
- enabling 7-4
- how it works 7-3

efpredict 10-5

Electrify 10-1

- arguments 10-2

eMake

- annotation 8-1
- command line options 3-8
- debug logging 11-2
- emulation 3-7
- fake interface for ClearCase 9-3
- invoking 3-2
- MAKEFLAG processing 5-15
- root directory 3-3
- variables 4-12

eMake root 3-3

emulation modes 3-7

environment variables

- configuring 3-5

G

gcc job caching 6-3

GNU Make

- unsupported options 5-2

H

hidden targets 5-11

history file 7-9

I

integration

- ClearCase 9-2
- Coverity 9-7
- Cygwin 9-8
- Eclipse 9-8

invoking eMake 3-2

J

Javadoc caching 6-16, 6-23

job caching, gcc 6-3

JobCache feature 6-3

L

ledger 7-6

local job 6-25

M

MAKEFLAG processing 5-15

multiple remakes 5-14

N

NMAKE

- inline file locations 5-15
- unsupported options 5-3

P

parse avoidance 6-17

performance optimization 6-2, 6-2, 6-3, 6-17

priority pools 4-5

proxy command 4-8

R

registry

  tools that access 3-5

S

sample build 3-24

serializing make instance jobs 6-26

shut down cluster host during builds 4-17

single make invocation 3-2

subbuilds 4-9

submake

  stub compatibility 5-8

  stubbed output 5-5

  stubs 5-6

T

temporary file

  deleting 6-29

  management 6-28

tools 3-4

  that access the registry 3-5

transactional command output 5-4

U

unsupported

  GNU Make options 5-2

  NMAKE options 5-3

V

virtualization 1-3

  environment variables 1-4

  registry 1-3

  user accounts 1-4

W

whole command-line matching 10-4

wildcard sort order 5-13