Vol.2018-IOT-40 No.46
2018/3/6

パッケージビルド基盤におけるキャッシングによる効率化

吉田 匠杜[1,a)]　鈴木 貢[1]

概要：オペレーティングシステムのディストリビューションにおけるパッケージ配布形態としてソースベースを採用する場合，資源が乏しい IoT 向けのターゲットでは，強力な計算機群を擁するビルド基盤を構成して，それにビルドの負担を引き受けさせることが考えられる．この場合，ソースコードやそれから生成される最終結果以外に多くの中間生成物が生成されるが，同じことの繰り返しになることが多いので，これをビルドツリー間で共有することで，資源消費や計算負荷を軽減させることができる．本論では，そのようなビルド基盤を提案し，その設計と実装について報告する．

Improving Cache Effeciency on Package Build Platform

Yoshida Tact[1,a)]　Suzuki Mitsugu[1]

1. はじめに

Unix 系 OS の草創期には，各ユーティリティはソースベースで配布され，これをユーザが調整しながらインストールしたいユーティリティをビルド（この頃は make）していた．しかし，ユーティリティ間でライブラリ等を共有するようになったり，複数のユーティリティの集合体が 1 つの機能を果たすようになると，パッケージという概念が生まれ，さらにパッケージ間の依存関係が定義されるようになり，必要とするパッケージをインストールすると自動的に依存するパッケージもインストールされるようになった．

パッケージの配布形態としては，Debian Linux 等のようにバイナリ形式のパッケージを主な配布の対象とするディストリビューションと，Gentoo Linux 等のようにソースコードを配布し，それを各ホストでビルドしてバイナリを得て，それをインストールするものがある．後者の利点としては，各ホストの構成に特化したコンパイラオプション等を適用したビルド結果をインストールすることにより，ホストを最適な条件で稼働できることにある．この利点は特に IoT 向けデバイスのような余裕が少ないハードウエアスペックでシステムを構成する場合に有効であるが，そのようなシステムでソースコードからインストールするものをビルドすることは不可能である．

そこで，パッケージのビルドを別の強力なホストに依頼し，バイナリ形式の最終生成物を自機にインストールすることが考えられる．本論ではホストがビルドを依頼するサーバの集合体をビルド基盤と呼ぶことにする．我々のゴールはこのようなサーバの集合体を構成する技術の開発であるが，本発表では単体のサーバによるビルド基盤の構成を報告する．

このように管理はホストごとに行い，作業はビルド基盤に委譲することにより，ホストの構成に特化した無駄のないシステムを構成するという本来の目的以外に，以下の事項を実現することを目標とする．

- コンテナや組み込み機器向け OS のチューニングを容易にする．
- パッケージのカバレッジの検証，ビルドログの収集やパッケージに含まれるファイルの検索を容易にする．

2. 関連研究

この節では本研究に関連する成果を見ていく．

本研究がベースにしている Gentoo Linix[1] は，Portage と呼ばれるパッケージ管理システムを採用しており，ソー

[1] 島根大学総合理工学部
Interdisciplinary Faculty of Science and Engineering, Shimane University. 1060 Nishikawatsu-cho, Matsue-city, Shimane 690–8504, Japan
[a)] s143112@matsu.shimane-u.ac.jp

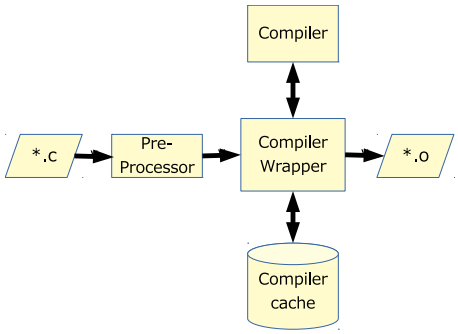

図 1　ccache の構成

スベースでパッケージの配信を行い，ユーザは自分の環境に最適な条件でシステムを構築することができる．この点は BSD 系 OS の ports に類似しているが，Gentoo ではより煩雑さが少なくなっている．ただし，IoT 端末のような非力なマシンで Gentoo が推奨しているような，システム構築方法，つまり，LiveCD 等でシステムを起動し，カーネル等の起動に最小限必要なバイナリをダウンロードし，Portage を使ってシステムを構築することは不可能である．本研究を思いつくに至った発端はこの点にある．

Hydra[2] は NixOS 向けの，多アーキテクチャ対応ビルドプラットホームで，ソースの蓄積からテストやリリースまでを連続的に行い，宣言的にビルド方法が記述可能で，かつ，ポータビリティがある．

Open Build Service(OBS)[3] は汎用のビルドプラットホームのためのキットで，openSUSE Linux を用いて構築されている．ccache や我々のような途中結果のキャッシングのための工夫は行っていない．また，OBS は開発者が配布する環境やパッケージの機能を決めてビルド成果を静的に配布するが，3.1 節で述べるように，GBS では利用者が環境やパッケージの機能をサーバーにリクエストしパッケージを動的に配布する点で異なる．

Arch Build System[4] は Arch Linux のためのビルドプラットホームで，BSD 系 OS の ports に類似している．これでも特にキャッシングによる負荷の低減は行われておらず，リモートビルドプラットホームを使うための手段はない．

ccache[5] は図 **1** のような構成を有する C や C++が生成するデータを管理するソフトウエア開発ツールである．コンパイルに関連するデータのハッシュ値を追跡し，コンパイル対象を必要なものに絞るので，2 回目以降のビルド時にコンパイルを回避して生成結果を再利用できるものはそのようにし，リビルドにかかる時間を劇的に減らすものである．ただし，対象が C と C++のみであるので，それ以外の言語には適用できない．本研究では，ccache がサポートしている言語に限りこれを活用している．

icecream[6] は分散コンパイル環境構築キットであり，コマンド `icecc` がコンパイル環境一式をスレーブに送り付けることで並列コンパイルを行う．このキットも C と C++のみにサポート対象が限定されている．

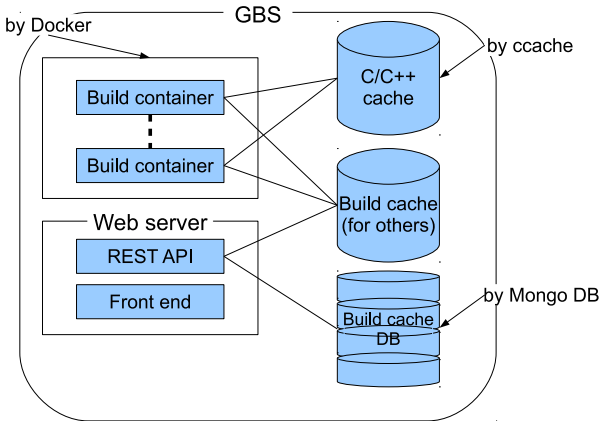

図 2　本研究で実装したビルド基盤の構成

distcc[7] も分散コンパイル用ツールであり，GCC をバックエンドとして利用している．distcc ではターゲットのアーキテクチャ毎にサーバを立ち上げる必要があり，途中結果のキャッシングも考慮していない．

kubernates[8] は Docker に対応したコンテナクラスタ管理ツールで，コンテナ化されたアプリケーションのデプロイやスケーリングを自動化することができる．今回の報告の内容では機能を十二分に活用してはいないが，我々が目指すビルドプラットホームを構築するために重要な役割を果たすと考えている．

SOURCEFORGE[9] や GitHub[10] は代表的なソースリポジトリである．これらはソースの受け入れと配布のみを行い，ビルドに関連する作業は行わない．

LLVM[11] は最適化コンパイラインフラストラクチャの 1 つであり，C や C++，それに Objective-C 言語対応のフロントエンドである Clang を備えている．我々のビルドプラットホームはこの言語処理系環境にも対応している．

3. 設計と実装

我々が実装したビルド基盤の概略を図 **2** に示す．この説ではそれぞれの項目について詳説していく．

3.1 GBS

本研究で開発した単一サーバによるビルド基盤を Gentoo Build Server(GBS) と呼ぶ．この節では，GBS の主な働きを説明する．

GBS では受け取ったビルドリクエストに含まれるパッケージの use flag やバージョン, 依存関係などを元にパッケージのビルドを Docker コンテナ上で行う．コンテナ上で行うことで kubernates などの分散プラットフォームの恩恵を享受できる．Docker コンテナには予めビルド用のスクリプトを配置しておき，コンテナ起動時に GBS から引数を受け取りビルドを開始する．リポジトリや成果物

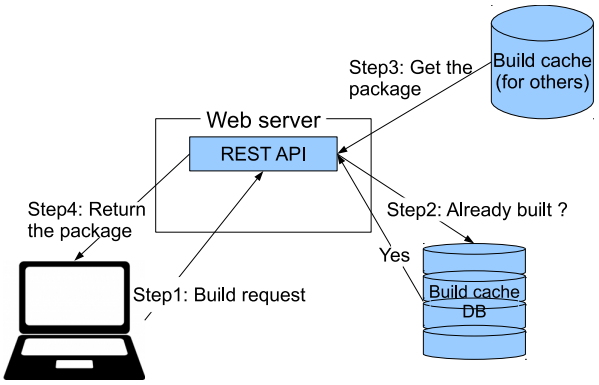

図 3　ビルドキャッシュがヒットした場合の GBS の動作

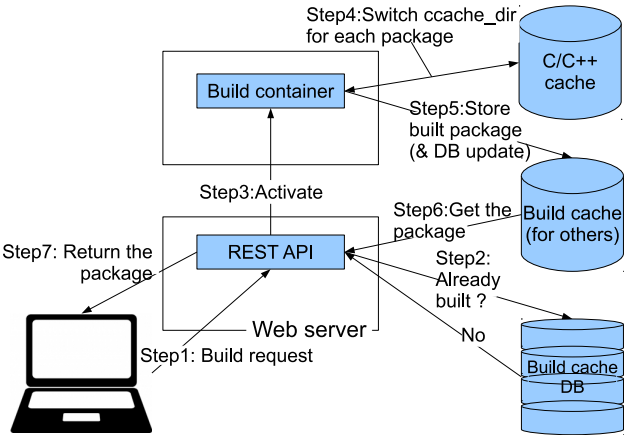

図 4　ビルドキャッシュがヒットしなかった場合の GBS の動作

(build cache) のディレクトリは，Docker コンテナの機能によりマウントする．ビルドリクエストが以前受けたものと同一である場合は，その際にビルドしたときの成果物 (build cache) を返す．

3.2　キャッシュ

この節では GBS におけるキャッシュについて説明する．

GBS ではパッケージレベルでのキャッシュ (build cache) と，ソースコードレベルのキャッシュを行っており，後者は ccache を用いて実現している．また，キャッシュを適用する順序は，build cache，ccache の順である．**図 3** にビルドキャッシュがヒットした場合の GBS の動作を，**図 4** にビルドキャッシュがヒットしなかった場合の GBS の動作を示す．

新しく追加されたパッケージ，バージョンアップが行われたパッケージなど，1 度もビルドされたことがないパッケージには ccache を適用しない．

3.2.1　パッケージレベルでのキャッシュ (build cache)

この説では build cache の動作について説明する．

build cache では use flag やパッケージのバージョン等が一致した場合に GBS のストレージにキャッシュされたビルドの成果物からバイナリパッケージ提供する．build cache がない場合は，個別に用意された ccache をマウントしてビルドを行う．

3.2.2　ccache によるソースレベルのキャッシュ

この節では ccache に関連する動作について説明する．

通常の ccache の利用方法では単一の ccache dir に全てパッケージのコンパイル結果をキャッシュとして書き込むので，容量制限が行われる場合は他のパッケージや使われていない，もしくは古いキャッシュは削除されてしまう．これは利用されるかもしれないキャッシュが削除される可能性があることを意味する．そこで複数の ccache dir を個別に管理することでこれを防ぎ，ソースコードレベルでのキャッシュの効果を高めている．

本研究の実装では，図 1 のような，ccache をパッケージのバージョンごとに作り，それぞれのパッケージのビルド前に ccache のディレクトリを切り替える．これによりパッケージ毎のキャッシュの一括削除を可能にした．

3.3　use flag

use flag は on/off を切り替えることでパッケージの機能などを選択できる Gentoo の機能である．パッケージごとにそれぞれ設定されており，共通のものは一括で切り替えることもできる．この機能により環境が必要とする最少の機能だけのパッケージを作ることができる．

3.4　ビルドリクエスト

ビルドリクエストとはパッケージ情報，ビルド環境，依存パッケージ情報の組である．パッケージ情報にはパッケージ名，カテゴリ，バージョン，リポジトリ名，use flag が記述されている．カテゴリはパッケージ名の名前空間のコリジョンを防ぐために存在している．リポジトリ名は Gentoo が公式に提供しているリポジトリと，非公式に提供されているリポジトリ (オーバーレイ) を区別するために用意されている．ビルド環境には CC，CXX，LD，libc，arch が記述されている．CC は C コンパイラ，CXX は C++コンパイラ，libc は glibc や musl など，arch はアーキテクチャを指定するために必要である．依存パッケージ情報にはビルド対象のパッケージが依存しているパッケージ群のバージョン，use flag が記述されている．依存パッケージのバージョンがホストによって異なることがあり，ライブラリ等のバージョンを合わせるために必要である．

4.　問題点と今後の課題

4.1　再帰的ビルドリクエスト

GBS ではビルドで依存関係として必要になったパッケージをコンテナから REST API にビルドキャッシュが存在するか問い合わせ，存在する場合はビルドキャッシュからビルド済みパッケージを取得する．そして存在しないか，ビルド中である場合は，コンテナ内でビルドを行う．本来であれば存在しない場合，あるいはビルド中の場合には，

表 1　実験を行った環境

OS	Gentoo/Linux
kernel	linux-4.11.0-rc4
CPU	i7-7820X
Memory	128GB
Docker	ver. 18.01.0-ce
gcc	ver. 6.4.0

表 2　ビルドの対象

	依存パッケージ数	1 回目の use flag	2 回目の use flag
emacs-25.3	5	acl, gtk3, inotify, xpm, zlib, ssl	acl, gtk3, inotify, xpm, zlib
coreutils-8.28-r1	0	acl nls xattr	acl nls
gcc-6.4.0-r1	0	cxx fortran nls nptl openmp pch sanitize ssp vtv	cxx fortran nptl openmp pch sanitize ssp vtv

表 3　実験結果（単位：秒）

	Emacs		coreutils		gcc(GCC)	
	通常	GBS	通常	GBS	通常	GBS
1 回目	96	134	73	87	1282	1320
2 回目	42	103	58	77	1260	1377
ビルドキャッシュヒット時	—	27	—	7	—	14

ビルドリクエストを投げるか, ビルドが終わるのを待つべきである．しかし本システムでは閉路検出を行っていないので, Portage から並列化できる依存関係を静的に取得してするようにすべきである．

4.2　依存関係解決サーバー

Gentoo の公式パッケージリポジトリは 1GB ほどあるため, ストレージの容量が少ない機器では, リポジトリの導入を行えない．また Portage には，インストールされたパッケージ数が増加すると依存関係の解決が遅くなる等の問題がある．依存関係解決用の API を提供することで，上記の問題を解決しつつ，真の PaaS(Package manager as a Service) を実現する．また，この API を用いることで，他の OS から Gentoo のパッケージを手軽に利用することができるようになる．

4.3　分散ビルド

今後, 複数のビルドホストで分散ビルドを行うために, kubernetes を用いて大量のビルドリクエストにも対応する．

5.　評価

GBS を用いてビルドした場合とそうでない場合で，ビルドにかかる時間の比較を行った．**表 1** に評価に用いた計算機環境を示す．make のジョブ数は 8 でビルドを行った．

実験に用いたビルドの対象を**表 2** に示す．1 回目と 2 回目でそれぞれ別の use flag を使いパッケージの構成を変更する状況を再現した．**表 3** に実験結果を示す．1 回目のビルドはキャッシュが全く効かない場合のビルド時間，2 回目のビルドは一部のファイルに ccache が効く場合のビルド時間となる．

「ビルドキャッシュヒット時」は既存のビルドキャッシュを利用できた場合の処理時間である．

Emacs のビルドでは，依存パッケージが多く，ビルドキャッシュのチェックのオーバヘッドや，他のビルドコンテナの負荷により，ビルド時間がかかっているものと考えられる．

coreutils のビルドでは，ccache が同じ程度に効果を発揮していることが分かった．

gcc などの中程度の負荷でも 100 秒以上の差がついたのは，ビルドコンテナ内部でインストールとバイナリパッケージ化の際の配置と圧縮が原因であると考えられる．

6.　結論

ビルドプラットフォームを構築したことで, 性能の低いマシンでも Gentoo を利用できるようになった．ビルドキャッシュがヒットした場合は，他のバイナリ配布型ディストリビューション遜色がない時間でパッケージを利用可能である．通常のビルド方式に比べて，今のところ提案方式は性能が劣っている．これは，システムの実現において何か落とし穴に嵌っているのではないかと考える．今後はこれや，4 節で述べた問題点を解決して，より実用性が高いビルドプラットフォームを構築していきたい．

7.　参考文献

参考文献

[1] ”Gentoo Linux”, https://www.gentoo.org（2018-1-30 取得）.
[2] ”Hydra: A Declarative Approach to Continuous Integration1”, http://citeseerx.ist.psu.edu/ viewdoc/download?doi=10.1.1.597.4843&rep=rep1&type=pdf（2018-1-30 取得）.
[3] ”Open Build Service”, http://openbuildservice.org/（2018-1-30 取得）.
[4] ”Arch Build System”, https://wiki.archlinux.org/ index.php/Arch_Build_System（2018-1-30 取得）.
[5] ”ccache — a fast C/C++ compiler cache”, https://ccache.samba.org/（2018-1-30 取得）.
[6] ”Distributed compiler with a central scheduler to share build load”, https://github.com/icecc/icemon（2018-1-30 取得）.
[7] ”distcc”, https://code.google.com/archive/p/distcc/（2018-1-30 取得）.
[8] ”Production-Grade Container Orchestration”,
[9] ”SOURCEFORGE dot net”, https://sourceforge.net/（2018-1-30 取得）.
[10] ”GitHub”, https://sourceforge.net（2018-1-30 取得）.
[11] ”The LLVM Compiler Infrastructure”, https://llvm.org/

（2018-1-30 取得）.