ソフトウェアビルド試験用のスクラッチ環境の高速化のためのコピー・オン・ライト・ファイルシステムの実装と適用

Implementation and deployment of copy-on-write filesystem for improving software build testing scratch environment system

上川 純一*
Junichi Uekawa

2007-09-14

概要

本論文では スクラッチ環境を提供し、ソフトウェアビルド試験を実施するための仕組み cowbuilder を紹介する。cowbuilder は Debian GNU/Linux およびその派生物に特化したソフトウェアビルド試験のフレームワークである pbuilder を応用し、ユーザ空間でのファイルシステムのコピー・オン・ライトを実装する cowdancer を実装・活用することで高速化した。また、ユーザ空間で拡張することで短期間な普及を実現した。本論文では cowbuilder の実装の仕組みと検討経緯を紹介する。

In this paper, 'cowbuilder', a scratch-test environment for Debian GNU/Linux, is introduced. It extends the existing Debian building/testing framework 'pbuilder', with 'cowdancer' to improve speed. Cowdancer is a copy-on-write filesystem implemented in user-space. Being fully implemented in user-space allowed for widespread adoption in a relatively short time. This paper explains the implementation of and discusses the design decisions for cowbuilder.

1 はじめに

Debian Project [1] の提供する Debian GNU/Linux は、10000 超[*1] のフリーソフトウェアパッケージを提供する大規模な Linux ディストリビューションである。このように大規模なソフトウェアの集合体を分散開発するためには、最終的な成果物の全体としてのシステムテストも重要になるが、全邸として、各モジュール単位でのテストを充実し、単体の信頼性を向上させることも重要になる。pbuilder [3, 4, 5] は Debian のモジュール単位であるソースパッケージ単位においてのソフトウェアビルド試験をする環境を提供する。pbuilder のねらいは簡便で標準化された手続きで利用できるテスト環境を提供することにより Debian GNU/Linux 全体の品質を底上げすることである。試験としては各種実施可能ではあるが、Debian GNU/Linux の全ソフトウェアで共通で実施可能な試験として特にソフトウェア自体のビルドが可能かであることを確認する試験を目的とする。全 Debian Developer の開発環境に導入、各個人で試験が実施できるようにすることを目標とするため、数千人程度の規模のユーザベース[*2] を想定している。この規模では、個

* Debian JP Project 会長、Debian Developer、
日本ヒューレット・パッカード株式会社 コンサルティング・インテグレーション統括本部

[*1] 2007 年 8 月時点現在のソースパッケージ数は 12250。[2]

[*2] 2007 年 8 月調査時点でメールアドレスを基準に数えるとパッケージを担当している Debian Developer は 749、スポンサーされている開発者は 1219 で、合計 1968。そ

別対応はほぼ不可能になる。そのため、設計の指針として、インストールしたらすぐに使える状態、もしくは一般的な手順書を提供してインストールできる程度の複雑度を目指している。

pbuilder が提供する試験環境は、Debian GNU/Linux を chroot 環境に仮想的にインストールし、chroot 環境を利用し試験をするものである。試験の再現性を担保するために環境は毎試行削除し作成しなおしている。chroot 用の試験環境ファイルシステムツリーを削除し作成しなおしていることを以後スクラッチ環境と呼ぶ。

Debian GNU/Linux の開発手順の標準文書として、Developers Reference[6] があり、pbuilder は標準的試験ツールとして文書化されている。pbuilder コマンドを利用すると、スクラッチ環境でのソフトウェアのソースコードからのビルド試験を `pbuilder build` **パッケージ名**で実施できる。ここで、スクラッチ環境を base.tgz という tar-ball から毎回展開して作成しなおし、テスト用のクリーンルーム環境を毎度再作成している。pbuilder には頻繁に実行する処理の一試行に関して数分程度の実行時間がかかる。待ち時間の開発者の生産性への影響を考えると、高速化できるなら高速化することに利益がある。

pbuilder において行われる代表的な処理は create、update、build、login である、その用途と頻度を表 1 に示す。その中で特に代表的な処理の実行時間内訳を分析した。update においてはアップグレードの処理 (upgrade)、および build においてはビルドに必要なアプリケーションのインストールやソフトウェアのビルド (build) が目的としている作業だが、実行時間のほとんどは目的としている作業ではなく、base.tgz の展開 (extract) や base.tgz の再作成 (recreate base.tgz) などの副次的な部分で消費されていることがわかった（図 1)。副次的な部分について別の方法を採用することでより高速に実現できるのであれば全体としての開発者の作業における待ち時間を削減できる。そのため、高速化の方策を検討した。高速化を実現するために cowbuilder を設計した。cowbuilder で利用するスクラッチ環境を実現するための一要素としてコピー・オン・ライトファイルシステムの実装方法を検討し、各種方式を検討した結果、ユーザ空間での実装である cowdancer を設計した。

表 1　pbuilder における代表的なコマンドとその実行頻度

操作	操作頻度	意味
create	最初に base.tgz を作成するときに一度	スクラッチ環境の初期作成
update	一日二回 (Debian unstable の新版のリリースにともなう更新)	スクラッチ環境 (base.tgz) を最新の状態にアップデート
build	開発者がパッケージビルドするたび	Debian パッケージをスクラッチ環境内でビルドする
login	問題解決の必要のある際に随時	スクラッチ環境にログインし、インタラクティブに操作する

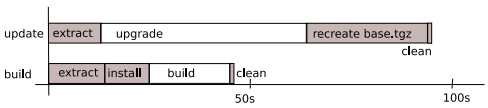

図 1　pbuilder update と build 処理の各処理内容の実行時間内訳（秒）

2　スクラッチ環境の既存の実装方法の pbuilder への適用の検討

pbuilder では、スクラッチ環境の確保のため、base.tgz を毎回展開・削除しているが、処理に時間がかかっている現状がある。スクラッチ環境の作成と削除を迅速に行うための方策の一つとして、ファイルシステム側に支援する仕組みを準備することが考えられる。例えば、テンプレートとなるファイルシステムツリーを用意しておき、書き込みは別の場

れに加えて登録されずに活動している開発者も存在していることを想定している。[2]

所にセッション中の間だけ保持するようにして終了すると最初の状態に迅速に戻る仕組みがあるとよい。その仕組みを実現するためにはいくつかの方式が考えられ、例を表 2 に示す。

表 2　各種スクラッチ環境の実装方式

種類	実装例
ブロックデバイス[a]	Device-mapper snaphot [b]
エミュレーション環境のファイルシステム [c]	user-mode-linux block device の COW 機能, qcow デバイス
ファイルシステム[d]	unionfs, aufs, funionfs
ユーザ空間[e]	fl-cow

[a] マウントしたブロックデバイスへの書き込みを別のファイルにリダイレクトする種類のもの
[b] LVM2 のスナップショット機能としても利用されている
[c] 通常の OS の範囲では利用できないが、システムエミュレータがエミュレーション環境で動作する OS が利用するディスクイメージに関して、スナップショット機能を提供しているもの
[d] 複数のファイルシステムを一箇所にマウントすることができ、書き込みを一部のファイルシステムに限定できるもの
[e] ハードリンクを活用することで、ツリーのコピーを作成し、アプリケーションから書き込みがある時点でハードリンクを破壊するもの

コピー・オン・ライトを実現するブロックデバイス [7] を利用する場合をまず検討する。device-mapper snapshot という仕組みを利用したものとして LVM2 のスナップショット機能が利用できる。しかしそれを準備するためには LVM2 用にディスク領域を確保する必要がある。この設定はまだ万人に普及しているとはいえないため、設定が簡便でなくなるという問題がある[*3]。エミュレーション環境のファイルシステムについては実際に user-mode-linux[8] と qemu[9] の実装を作成した[*4]。しかしながら user-mode-linux と qemu では、エミュレーション環境によるオーバヘッドと各種操作性への影響が、スナップショットファイルシステムの利便性によってもたらされる利便性よりも大きく、user-mode-linux の提供する機能である一般ユーザ権限での実行などが必要な場合や、qemu の提供するエミュレーション環境が必要な特殊用途への活用はみられたが一般的に普及するには至らなかった。一例として、 qemu を利用した場合の実行時間の例を図 2 に示す。qemu での実装では、COW ブロックデバイスの実装を活用しているため、スクラッチ環境の準備に時間がかかってはいないが、そこで節約されている時間以上に仮想マシンの起動 (OS boot) に時間がかかっており、またビルド処理などはエミュレーション環境で実施しているため時間的なオーバヘッドが大きいことが確認できる。

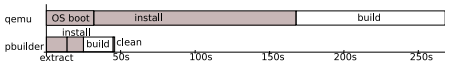

図 2　pbuilder と qemubuilder の build 処理内容の実行時間内訳（秒）

また、コピー・オン・ライトを実現するカーネルレベルで提供されるファイルシステム [7] については決定的なものがなく、現在においても Linux カーネルのメインラインに採用されるに至っていない。unionfs[10]、aufs[11] などについてはまだ活発に開発されており、どういう方法が最終的にエンドユーザのシステムに導入するのが好ましいのかのという判断はまだしにくい。標準で提供されているものがないため、インストールにはカーネルのビルド等の作業が必要になる可能性があり、それをユーザに要求するのは好ましくない。そのため、広範に普及させるシステムに採用するのは難しい。fl-cow[12] については、pbuilder では root 権限での操作が大半になるのに対して、 fl-cow は一般ユーザ権限での稼働を前提として設計されており、用途に対して機

[*3] 今後もし LVM2 が Debian GNU/Linux のデフォルトインストールでの標準設定となれば数年後には普及している可能性もあるが現時点ではデフォルトでもなく、広く普及もしていない。

[*4] pbuilder-uml[5] と qemubuilder[4, 3] が実装されている。

能が十分でないため採用できなかった。

fuse[13] による実装 funionfs[14] については今回は検討しなかった。fuse にはユーザ空間とカーネル空間を組み合わせた実装であることなどによる各種特性があることを想定した。特に、fuse は 一般ユーザ権限での利便性を主として開発を進めている傾向があり、root 権限での利用を想定している今回の用途にはあわない可能性があった。また、 fuse が Linux カーネルのメインラインに採用されたのは比較的最近であるため、他の各種ファイルシステムと同様に普及する際の制限が検討当時はあった。funionfs がソフトウェアビルド試験を実施するための環境として適切であるのかを判断する段階にないとみて今回は検討を見送っている。

以上の理由より新規開発する方針を採用した。

3 コピー・オン・ライトをユーザ空間で実現する方式の検討と実装

3.1 ユーザ空間実装の意義

狭義のファイルシステムは通常カーネル空間で実装されているものを指す。ファイルシステムとして提供されている機能として等価なものをユーザ空間で実装することができれば、カーネルに変更を加えないで利用できるファイルシステムができる。

ユーザから見ると、カーネルに手を入れることは安定性にかける、特にあまり広く使われていないカーネルモジュールを導入することは、使い込まれていないことによる安定性に対するリスクを及ぼす。また、ユーザのインストールの手間の観点で好ましくない。pbuilder の対象とするユーザ、例えば Debian Developer の中ではカスタムカーネルを利用しているメンバーが多いと想定される。そのため、カーネルモジュールのコンパイル処理にかかる時間、および各種カーネルバージョンに対応するための多くの組み合わせについて考慮が必要になる。特に動作可能な組み合わせが多くなることは、問題が多く発生することにつながりやすい。また、カーネルを操作する場合に、再起動が必要な操作になる可能性がある。

ユーザ空間のみで実装する方法を採用すると決定した場合にも、各種の方式が検討できる。設計について以下で詳細を説明する。

3.2 ユーザ空間の各種方式のメリット・デメリットの検討

UNIX にはハードリンクという機能がある。ハードリンクでツリー全体を複製すると、読み込んでいるだけで書き込まない限りは有用なファイルシステムができる。書き込みはツリーの複製元にも影響をあたえるので、変更対象のファイルを書き込み直前に複製に置き換える処理（コピー・オン・ライト）を実行することであたかもすべてをコピーしたかのような状態を準備できる。この仕組みはスクラッチ環境の実装に好ましい。ユーザ空間でこの仕組みを実装をするためには、アプリケーションが書き込みアクセスをしようとする直前に検出し、アプリケーションを一旦停止、対象ファイルのコピーを作成し、その後アプリケーションの処理をコピーに対して再開するための方法が必要になる。

Linux システムで上記の方策を実現する方法としては、ptrace[15]、LD_PRELOAD[16]、inotify[17]、dnotify[18] などが考えられる。それらが今回の用途に適しているのか、実装する上でどれくらいの困難がありそうかを評価した。

まず、アプリケーションによるファイルの書き込みを通知する仕組みとしては Linux カーネルの提供する inotify、dnotify がある。しかし、調査した限りではファイルの変更直前にアプリケーションの動作を停止するという挙動を想定していない仕組みのため、今回の用途には活用できないようだった。

次に ptrace と LD_PRELOAD で実装することのメリットとデメリットを検討した。比較した結果を表 3 に示す。ptrace と LD_PRELOAD を比較した結果、ptrace のアーキテクチャ毎の実装度合が異なる点、および fork を正確に扱うためには実行時のコード書き換えなどの技が必要になるため、実装が複雑になるだろうことが判明した。複雑な実装になることは cowdancer ソフトウェアの維持管理コストに影響することが想定される。そのため、今回は LD_PRELOAD にて実装することにした。

表 3　ptrace と LD_PRELOAD の機能比較

	LD_PRELOAD	ptrace
システムコールのパラメータの取得形式	関数のパラメータ	CPU レジスタ[a]
open が呼ばれた状態の追跡	可能	可能
open システムコールをアプリケーションコードが直接呼んだ場合の追跡	不可	可能
suid アプリケーションの追跡 (例：gpg)	不可	不可
static link アプリケーションの追跡	不可	可能
アプリケーション側からのトレースからの回避方法	可能[b]	不可能
fork された場合の処理継続	継続	非継続[c]
chroot した場合の対応	可能[d]	可能
fakeroot との互換	可能	可能
アプリケーションのシステムコールを無効にするため、システムコールを発行させない	可能[e]	不可能[f]
利用実装例	fakeroot, fakechroot, auto-apt	gdb, user-mode-linux, strace, ltrace

[a] どのレジスタにどの値が入っているのかは CPU アーキテクチャとそのアーキテクチャでのカーネルのシステムコール規約に依存する
[b] LD_PRELOAD 環境変数のリセットなど (例: ldd)
[c] attach しなおすことで実現可能。空白の時間が発生しないようにコードを動的に書き換えるなどの、実装側での工夫が必要
[d] LD_PRELOAD のパスの指定先が chroot 内部でになる
[e] システムコールを呼ばなければよい
[f] user-mode-linux は、影響を回避するために getpid() に変換してしまうことで実装

3.3 実装の方法

まずハードリンクされている保護対象のファイルに影響するような書き込みが発生する場合をトラップする必要がある。UNIX でハードリンク先に影響を与える変更が発生する契機となるシステムコールは、ファイルのオープン (open) と、i-node のメタデータの変更 (chmod 等) である。

一般的なアプリケーションは標準 C ライブラリ経由でカーネルの機能を利用する。Debian GNU/Linux の場合は GNU Libc[19] (glibc) 経由で Linux カーネルのシステムコールを呼び出す。コピー・オン・ライトの仕組みを LD_PRELOAD で実装するには、まずファイル書き込みを発生させる変更を発生するシステムコールを発生させそうな glibc の関数を選定する。そして、アプリケーションがそれらの glibc 関数を呼び出す場合に処理を奪い glibc の代わりに処理する仕組みを実装すればよい。

LD_PRELOAD 環境変数で指定すると ld.so（ダイナミックリンカ）はアプリケーションをロードする際に指定した共有ライブラリを利用する。その共有ライブラリが glibc と同様のシンボルを提供していると、アプリケーションは優先的に LD_PRELOAD でロードされているほうのシンボルを利用する。

cowdancer では、その仕組みを利用するため、LD_PRELOAD で指定するための libcowdancer.so 共有ライブラリを準備した。libcowdancer.so は、書き込みアクセスがある際には対象となる i-node を確認し、保護の対象でありハードリンクカウントが 1 以上であるならコピーしてから書き込みアクセスを行うような各種関数を提供する。ユーザが利用しやすいように cow-shell というラッパースクリプトを準備した。cow-shell は LD_PRELOAD の値を libcowdancer.so に設定してからシェルを呼び出す。また、cow-shell は保護対象となる i-node の一覧を初期化作業として作成し、ファイルでキャッシュするようにした。

3.4 制限事項

今回の方策での制限事項を整理する。

LD_PRELOAD を利用するため、スタティックリンクのアプリケーションには対応できない。また、glibc を経由せずに直接システムコールを発行しているアプリケーションには対応できない。

ハードリンクで情報を管理をするため、もともとハードリンクをされていた場合でも別のファイルとして分離されてしまう。この挙動は正確ではない。もともとのハードリンクの情報を管理することで対処が可能だが、管理のためのオーバヘッドが大きくなるため実装していない。

以上の制限事項については、Debian のインストール及び apt のパッケージインストールの部分が正常に稼働すればよいという前提のもとでは大きな問題となっていない。

3.5 速度の検討

LD_PRELOAD を利用するため、 cowdancer は初期化をアプリケーションのロードのたびに毎回実施することになる。また、 cow-shell コマンドは、保護対象のファイルを確認するため、対象ディレクトリ以下を全検索する。oprofile[20] で実行時のプロファイルを取得して確認したところ、主となるオーバヘッドはファイルの検索ではなくアプリケーションがシステムコールを呼び出すたびに保護対象の i-node であるかどうかを確認する部分だということがわかった。特にメモリ検索で保護対象の i-node 情報を全検索していた部分を、 cow-shell による初期化時点でソートしシステムコールの際には二分探索するように変更したところ高速になった。[21]

3.6 エミュレーション精度の検討

エミュレーションの目標としては、Debian GNU/Linux の chroot 環境の再利用ができる程度のレベルに設定した。すなわち、全てのアプリケーションが稼働できる必要はなく、 apt, dpkg, および、dpkg の postinst スクリプト等で動作するもののみが正常にエミュレーションできれば十分である。動作確認として実際にインストール・動作試験を行い、抜き打ち試験を行った。

システムコールの一部を glibc 経由で呼び出す段階で把握する実装のため、全ての書き込み可能な処理を網羅できる実装にはなっていない。不具合が起きたらその都度バグレポートとして手動で報告され、随時修正される予定である。バグ報告をベースに修正された例として、GNU Screen が正常に動かない、というバグ報告があった[*5]。調査した結果、これは configure スクリプトが close したあと open をしており、open の結果として最初 close したファイルディスクリプタが利用できることを想定して設計されていたのだが、 cowdancer 環境では close したのと違うファイルデスクリプタがかえされたため、configure スクリプトが正常に動作しないという問題だった。発見された問題については cowdancer のユニットテストに追加し、再発しないようにしている。

cowdancer でカバーしている glibc の関数のラッパーを表 4 に示す。cowdancer の開発期間中にもシステムコールが追加され glibc に関数が追加されている。新規の関数をアプリケーションが呼び出すようになった場合、 cowdancer 側での対応が必要になる。

大きな課題になるのは、挙動が一貫していないことによる例外処理の多さだ。glibc は Linux カーネル用に限ってみても、アーキテクチャ毎に異なるソースコードから生成されており、アーキテクチャ毎に若干挙動が違う。たとえば、i386 では、過去の Linux カーネルの挙動との互換性を維持するため、glibc 2.1 以降と以前とで chown 関数の実装が切り替わるようになっている[*6]が、amd64 ではそもそも glibc 2.1 以前が存在しなかったのでそうなっていない。そのため、例えば amd64 と i386 の glibc ではバージョンシンボルを指定しない dlsym で chown 関数を求めた場合のデフォルトの挙動が異なるという状況になる。

[*5] http://bugs.debian.org/413912

[*6] 昔の chown は現在の lchown と同様の挙動だった

表 4　実装した glibc 関数のラッパー

対応関数	備考
open	
open64	
creat	
creat64	
fopen	
fopen64	
chown	一部では GLIBC_2.1 バージョンが必要
fchown	処理せず警告を出す
lchown	
chmod	
fchmod	処理せず警告を出す

3.7　cowbuilder の普及

結果としての cowdancer の 普及度合について確認した。Debian GNU/Linux のパッケージインストール状況は、popcon[22] というシステムでトラッキングしている。このシステムは全数検査ではないため、完全な情報は得られない。しかし、一部の抽出調査の結果として利用し、ユーザ数の最小数の推察するために利用できる。ここでは、cowbuilder ツールの普及度合の検討に応用できると考えられる。

図 3 にグラフを示す。2006 年 5 月に cowbuilder が実用に耐える状態で投入されてから増加が見え、2007 年 7 月時点で 600 票が見られる。cowbuilder は pbuilder を必ずインストールすることになっており、現在の pbuilder は 1600 票程度である。cowbuilder の増加の勢いが大きいため、pbuilder を利用している層への cowbuilder の普及が進んでいるだろうと考えられ、また pbuilder ツールの利用も促進されていることが推察できる。

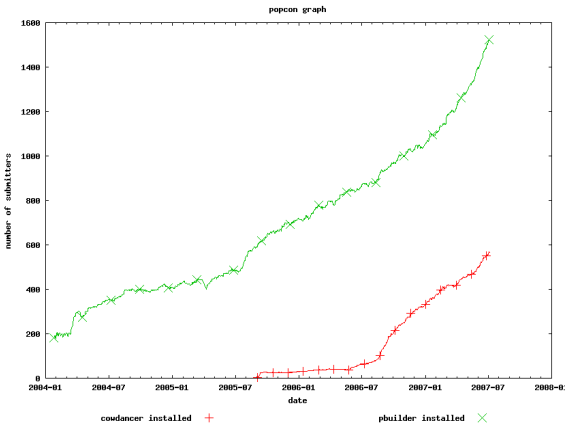

図 3　pbuilder と cowdancer の利用数

cowbuilder も pbuilder もは現在 Debian の正式パッケージとして提供されており、それぞれ `apt-get install cowdancer` 、`apt-get install pbuilder` でインストールできる。

4　pbuilder と cowbuilder の実効速度比較

4.1　検証環境

cowbuilder が pbuilder と比較してパフォーマンス向上をもたらしていることを確認するために比較検討した。検証環境として HP dx5150 (CPU: AMD Athlon64 3500+(2.2GHz)、メモリ: 1GB)、ストレージデバイスとして内蔵ハードディスク (250GB) を利用した。ファイルシステムは ext3 で、デフォルトの値で設定している。Linux カーネルは 2.6.23-rc3 のカスタムビルドを利用した。ソフトウェアビルド試験の例題としては、autoconf/automake (`./configure`) を利用する一般的な構成で小さいパッケージの例として dsh ソースパッケージ `dsh_0.25.9-1.dsc` を利用した。./configure スクリプトは GNU 標準として一般的に使われている方式で、 多数の小さな C 言語のプログラムを実際にコンパイル・実行することでコンパイル・実行環境の確認を行う挙動から、頻繁に書き込み・実行などの処理が行われる。その特性よりシステムコールのラッパー等を実装する際にはパフォーマンスの影響が出やすいと考えられる。

pbuilder と cowbuilder のもっとも頻繁に利用さ

れる処理である update 処理・build 処理[*7]・login 処理の時間を計測した。依存関係のインストールとアプリケーションのビルドに必要な時間の影響を排除・確認するため、ネットワークを切断した状態での build テストも実施した。

4.2 比較検討結果

計測結果を表 5 に示す。cowdancer がシステムコール関連のラッパーとして追加処理を行っているため、処理にはオーバヘッドが予想される。特に build の処理などに影響が出ることが想定できる。測定した結果、影響はほとんどなく、全体として速度の向上が見られることが確認できた。pbuilder と cowbuilder を利用した update, および build の場合のそれぞれの時間の配分を図 4 に図示する。

表 5　単純操作にかかる時間の比較（秒）

操作	pbuilder	cowbuilder	速度向上比
update	99	14	7x
build (N/W down)	31	5	6x
build (dsh)	67	52	1.3x
login	28	4	7x

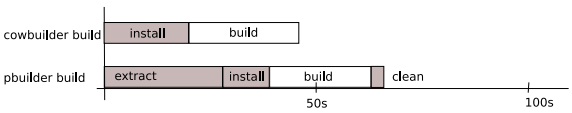

図 4　pbuilder build と cowbuilder build の実行時間内訳の比較（秒）

4.3 cowbuilder の今後の改善の可能性

cowbuilder は現状の実装で pbuilder にくらべ十分高速になっており、利用者の反応もよい。cowbuilder のパフォーマンスは特に問題になってはいない。ただ、今後の拡張などでもし cowbuilder のさらなる高速化が必要になるとすれば Linux カーネルなどのレイヤで改善することを検討している。LD_PRELOAD でのユーザ空間の実装に起因するオーバヘッドとハードリンクを流用している制限があることに着目すると、カーネル空間の実装で解決できる問題であると想定している。また、ファイル数が大きくなってきた場合のパフォーマンス劣化がおきた場合の対処については、Linux カーネルの ext3 ファイルシステムのファイルの削除処理についての高速化と最初にハードリンクの作成する部分の高速化を検討したい。

また、 cowdancer は設計上 glibc の変化に影響を受けやすいため、今後の glibc の変更とともに改修が必要になることを想定している。

[*7] 試験対象は pbuilder パッケージのビルド

5 まとめ

テストツール pbuilder のパフォーマンス面での課題を解決するためのツール cowbuilder を紹介した。また、 cowbuilder の主要部品として、 cowdancer の実装を紹介した。cowdancer はすべてのケースに対応できる実装ではないが、本論文で対象とした領域に関しては実用的な機能と速度改善を提供している。

参考文献

[1] Debian Project. *Debian.* http://www.debian.org/, 2007.

[2] Gürkan Sengün. *Status of Maintainer's packages, ports and bugs.* http://io.debian.net/~tar/bugstats/.

[3] Junichi Uekawa. discussing cowbuilder / pbuilder / pbuilder-uml / pbuilder-qemu. *Debconf7 Proceedings*, pp. 32–34, 2007.

[4] Junichi Uekawa. *pbuilder users manual.* http://pbuilder.alioth.debian.org/, 2007.

[5] Junichi Uekawa. *pbuilder.* http://www.netfort.gr.jp/~dancer/column/2004-debconf4.html.en, 2004.

[6] Andreas Barth, Adam Di Carlo, Raphaël Hertzog, and Christian Schwarz. *Debian Developer's Reference.*

http://www.debian.org/doc/developers-reference/, 1997-2007.

[7] 丹英之, 須崎有康, 飯島賢吾, 八木豊志樹. Copy on write を用いたオーバーレイブロックデバイスの実装. *Linux Conference 2005*, 2005.

[8] Jeff Dike. *The User-mode Linux Kernel Home Page*. http://fabrice.bellard.free.fr/qemu/.

[9] Fabrice Bellard. *QEMU*. http://fabrice.bellard.free.fr/qemu/.

[10] Charles Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, Erez Zadok, and Mohammad Nayyer Zubair. Versatility and unix semantics in a fan-out unification file system. *Stony Brook U. CS TechReport FSL-04-01b*, 2004.

[11] Junjiro Okajima. *Aufs – Another Unionfs*. http://aufs.sourceforge.net/.

[12] Davide Libenzi. *fl-cow*. http://xmailserver.org/flcow.html, 2005.

[13] Miklos Szeredi. *Filesystem in Userspace*. http://fuse.sourceforge.net//, 2007.

[14] Stephane Apiou. *FunionFS*. http://funionfs.apiou.org/.

[15] *Linux manual page: ptrace*.

[16] *Linux manual page: ld.so*.

[17] Robert Love. *inotify, a powerful yet simple file change notification system*. linux-2.6/Documentation/filesystems/inotify.txt, 2005.

[18] Stephen Rothwell. *Linux Directory Notification*. linux-2.6/Documentation/dnotify.txt, 2000.

[19] Free Software Foundation. *GNU C Library*. http://www.gnu.org/software/libc/.

[20] John Levon. *OPROFILE*. http://oprofile.sourceforge.net/news/, 2007.

[21] 上川純一. 第 57 回カーネル読書会：powerpc linux で oprofile. http://www.netfort.gr.jp/~dancer/diary/junk2006/20060117-ppcoprofile.pdf, 2006.

[22] Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. *Debian Popularity Contest*. http://popcon.debian.org/, 2004-2005.