カーネルウェア：アプリケーションプログラムのカーネル内実行によるOS機能拡張法の提案

佐藤 喬[†] 安田絹子[††]
中村嘉志[†††] 多田好克[†]

本論文では，アプリケーションプログラムを変更することなく汎用OSのカーネル内で実行し，OS機能を拡張する手法を提案する．ユーザモードで動作するアプリケーションプログラムには保護境界をまたぐ際にコンテクスト保存などのオーバヘッドが存在する．このことが，処理の性能を著しく低下させる1つの原因となっている．この問題に対し，特定のアプリケーションの機能をカーネル内に実装することでオーバヘッドを削減し，その性能を改善するという試みがなされてきている．カーネルモジュールでの実装はアプリケーションプログラムのソースが再利用できず，再実装が必要となり開発の効率が悪い．もし，アプリケーションを改変することなしにカーネル内実行できれば開発効率は格段に向上するであろう．そこで我々は，ソースコードの再利用性に着目し，アプリケーションプログラムをそのままカーネル内実行することのできる機構を実現した．本機構上でカーネル内実行されるアプリケーションプログラムをカーネルウェアと呼ぶ．既存のアプリケーションプログラムをカーネルウェア化することで，容易にアプリケーションプログラムの機能をカーネル内に実装し，改良することができる．システム利用例として，ls と cp コマンドのカーネルウェア化を行った．その結果，ソース変更なしの ls コマンドで約5%，ソース変更ありの cp コマンドで最大約26%の処理時間の削減が観測された．

Kernelware: OS Extension Mechanism by Executing Application Programs in Kernel

Takashi Satou,[†] Kinuko Yasuda,[††] Yoshiyuki Nakamura[†††] and Yoshikatsu Tada[†]

Kernelware is an application program, which runs in kernel without any modification. Developers of the kernelware have a chance to improve their application programs using direct I/O since the application program in kernel can access kernel resources directly. The developers can concentrate on such improvements because the other parts of the sources can be left unchanged. In this paper, we present the design and implementation of our system, and show examples of accelerating the ls and cp commands using on our system. We unchanged the ls command to run on our system. We modified the cp command to call a function that copies buffer caches directly between files. In our experiment, the execution time of the ls command was reduced by up to 5%, and the execution time of the cp command was reduced by up to 26%.

1. 導　　入

近年，Webサーバやデータベースサーバといった大規模なアプリケーションプログラム（以下AP）の活躍の場が増えてきている．これらのAPは大量のI/O処理を必要とする．I/O処理を行う場合，APはOSの保護境界をまたぎカーネルモードへ遷移する必要がある．このモード遷移はコンテクストの保存や復帰といったオーバヘッドの大きな処理をともなう．そのため，頻繁なI/O処理を行うAPは著しく性能が低下してしまう[1)]．

この問題を解決するため，APが行うサービスをカーネル内で提供する手法がある[1),2)]．カーネル内であれば，モード遷移のオーバヘッドを受けることなくI/O

† 電気通信大学大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications
†† 慶應義塾大学 SFC 研究所
Keio Research Institute at SFC
††† 産業技術総合研究所情報技術研究部門
Information Technology Research Institute, National Institute of Advanced Industrial Science and Technology

処理が可能になる．さらに，カーネル内の低レベル操作を使い，より効率的な I/O 処理を実現できる．

しかし，このような解決策は開発効率が悪いという問題がある．なぜなら，カーネル内で実行されるコードの記述には AP のソースをそのまま再利用することができず，多くの変更が必要となるためである．これは，カーネル内では AP の利用しているインタフェースやセマンティクスが利用できないことが原因である．

そこで我々は，カーネル内で AP の利用するインタフェースとセマンティクスを提供し，AP に変更を加えることなくカーネル内で実行する機構を実現した．もし，AP を変更することなしにカーネル内で実行する仕組みをつくれば，運用実績の豊富な既存の AP のサービスを容易にカーネル内の機能として提供でき，開発効率の問題を解決できる．

本機構を使ってカーネル内で実行される AP をカーネルウェアと定義する．AP をカーネルウェア化することで，I/O 処理におけるモード遷移のオーバヘッドを削減できる．さらに，カーネルウェアからカーネル内資源を直接操作できる枠組みを用意し，システムコールでは実現不可能な，AP の処理内容に最適化した I/O 処理手段を開発者に提供する．

本機構は汎用 OS の 1 つである NetBSD 上で実装した．そのため，動作実績のある AP が豊富に存在する．開発者はそれらの既存 AP をカーネルウェア化することで，容易にカーネル内に取り込むことができ，効率的なサービスを提供できる．

2. カーネルウェア化の提案

本機構を使ってカーネル内で実行される AP をカーネルウェアと呼ぶ．本機構はカーネルウェアに対して，ユーザモードで動作するのと同じインタフェースとセマンティクスを提供する．これにより，AP に変更を加えなくてもカーネル内実行を可能にする．さらに，カーネルウェアからカーネル内資源を直接操作できる枠組みを用意する．これにより，カーネルウェアの処理内容に特化した I/O 処理手段を開発者に提供する．本機構を使うことで，Web サーバやデータベースサーバといった大量の I/O 処理が必要な AP をカーネルウェア化し，効率的な AP サービスを実現する．

本機構を用いて AP のカーネルウェア化を行う場合の流れを説明する．行う手順は大きく 2 段階に分けられる．

まず，開発者はカーネルウェア化したい AP を用意し，本機構を用いて**図 1** のようにカーネル内実行する．本機構は，AP に対してユーザモードで動作するのと同じインタフェースとセマンティクスを提供する．そのため，AP へ変更を加えることなくカーネル内実行が実現できる．これにより，AP はモード遷移のオーバヘッドを削減することができ，その分のシステムコールの効率化が可能となる．

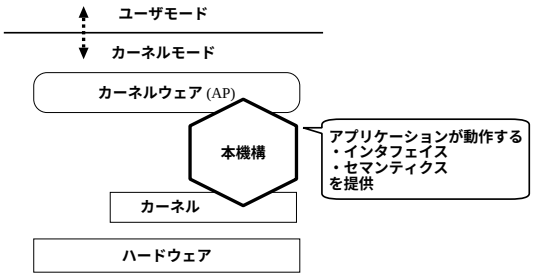

図 1 アプリケーションプログラム（AP）のカーネル内実行
Fig. 1 Execution of an application program in kernel.

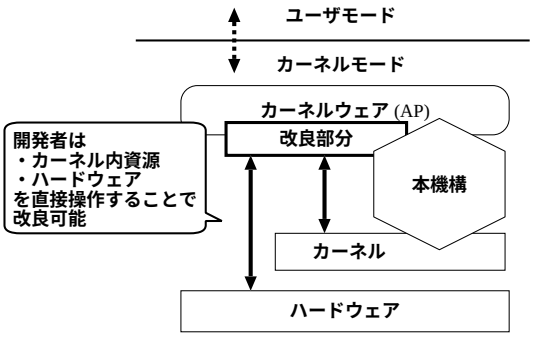

図 2 計算機資源の直接利用による改良
Fig. 2 Improvement using direct I/O processing.

次に，開発者は AP の処理内容に応じて，**図 2** のように計算機資源を直接操作する改良を加え，I/O 処理のボトルネック部分を解消する．この際，改良箇所以外のソースは元の AP のものをそのまま流用できる．これにより，開発者は改良箇所にのみ集中すればよい．そのため，すべてを書き直す必要があるカーネルモジュールに比べ，格段に少ない手間で効率の良い開発ができる．

例として，既存 Web サーバをカーネルウェア化する場合を考える．開発者は，ユーザモードで動作する既存 Web サーバを本機構によりカーネルウェア化する．Web サーバはユーザモードと同じ動作をしながら，モード遷移のオーバヘッドを解消できる．もし，モード遷移を省いたことで十分な性能が得られればそこで開発を終了する．そうでなければ，さらなる I/O 処理ボトルネックの解消を行う．たとえば，ボトルネック部分がファイル転送にあると判断すれば，転送データが存在するバッファキャッシュの内容を直接 NIC に

転送し，データコピーの回数を減らすことでI/O処理のボトルネックを解消するといったことができる．このように，ユーザモードで動作するAPでは難しい，カーネル内ならではの改良が可能である．

既存APを再利用することで，カーネルモジュールよりも開発効率を高めながら，カーネルモジュールと同様の計算機資源を直接操作する改良を加えることができる．ルータなどの最近の組み込みシステムでは，汎用OS上で少数の特定APを動作させていることが多い．このような場面において，本機構を使い特定APに的をしぼって改良するアプローチは有効である．

3. 設　　計

ここでは，本機構の設計方針と実現にあたっての要求点について述べる．

3.1 方　　針

開発効率が高く，拡張の容易なカーネルウェアを実現するため，本機構では以下のような設計方針をたてた．

- APの変更は最小限：
  APの再利用性を高めるため，APに必要な変更は少ないことが望ましい．APを再コンパイルしなくてもカーネル内実行できることが理想である．
- カーネルウェアから計算機資源が直接操作可能：
  I/O処理の効率化を行うには計算機資源，つまりカーネル内資源を直接扱えることが必要である．そのために，カーネル内の関数や変数をカーネルウェア内から扱えなければならない．
- OSや他のAPへの影響は最小限：
  本機構を導入したことによって，OSや関係ない他のAPへ影響を与えることは望ましくない．
- OSは広く利用されているものを使用：
  広く利用されているOSには，その上で動作する豊富なAPが存在する．APが豊富にあれば，本システムの利用局面を増やすことができる．

3.2 要 求 点

本機構はカーネルウェアに対し，ユーザモード動作と同じインタフェースとセマンティクスを提供し，かつ，カーネル内実行を利用した拡張性も提供する．そのための要求点を以下にあげる．

- カーネル内実行環境の提供：
  APをカーネル内で実行する機構が必要である．その際，APの変更は最小限に抑える．また，カーネルウェア中の一部の処理のみをカーネル内実行できるように，開発者側で動作モードの切替えを可能にするインタフェースを提供する．これにより，I/O処理のオーバヘッドが大きな部分のみをカーネル内実行し，オーバヘッドを削減できる．
- システムコールの関数呼び出し化：
  カーネルウェアは，ソフトウェア割込みを使った通常のシステムコールを使わなくてもよい．そこで，呼び出しオーバヘッドの少ない関数呼び出し型のシステムコールを提供する．このシステムコールを使うことで，開発者はAPに変更を加えなくても性能向上を実現できる．
- カーネル内シンボルの解決：
  カーネル内の関数や変数をカーネルウェアから扱うには，それらのシンボルとアドレスの対応を解決する仕組みが必要である．これにより，カーネルウェアからカーネル内の資源を直接操作し，処理内容に応じた改良をすることができる．
- プリエンプティブ：
  カーネルウェアは，ユーザモードで動作するときと同様にプリエンプティブであり，処理を占有してはいけない．そのため，割込みが発生した際，必要ならば他のAPへ処理を明け渡す必要がある．
- 安全性：
  カーネルウェアが異常動作をした場合，OSに対して悪影響を与える可能性がある．そのため，安全性に対して対策をとらなくてはならない．ただし，最近のOSは，個人用，もしくは専用サーバとして使われることが多い．そのため，本システムでは，保護の厳格さよりも，導入のコストや実行時のオーバヘッドを抑えることに重点を置く．

4. 実　　装

本システムはNetBSD–1.5.3/i386上に実装した．NetBSDはUNIX系のOSであり，その上では豊富なAPが動作している．カーネルに加えた変更はすべてLKM（Loadable Kernel Modules）により実現した．そのため，本システムの導入は動的に行え，カーネルの再コンパイルは必要ない．LKMによりカーネルに組み込まれるモジュールは，C言語で約2,000行，アセンブリ言語で約800行で記述されている．

4.1 カーネル内実行環境

実行の開始は，APを実行するexecシステムコールをフックし，実行開始のコードセグメントをユーザからカーネルへ変更することで実現する．このexecのフックは特定プロセスのみシステムコールテーブルを書き換えることで行う．そのため，カーネルウェア以外のAPに対しては影響がない．

フックしたexecシステムコールを使用することで，

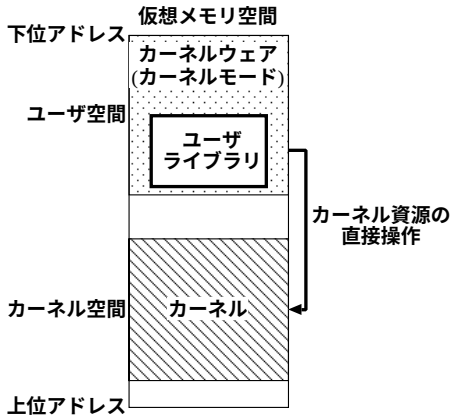

図 3 カーネル内実行時のメモリ配置
Fig. 3 A memory layout of a kernelware.

カーネルウェアは**図 3** のようにユーザモードで動作する AP と同じメモリ配置になる．そのため，AP へ加える変更を抑えることができる．

カーネルウェアは，カーネルモードで実行可能な点を除いて，ユーザモードで動作するプロセスと同じように管理されている．そのため，fork や exec などのプロセスの生成や実行をするシステムコールも利用可能である．ただし，これらのシステムコールは CPU の動作モードをユーザモードに戻してしまう．そのため，本システムではこれらのシステムコールをフックして，カーネルモードに書き換える処理をしている．

また，カーネルウェア中の I/O 処理オーバヘッドの大きな処理のみをカーネル内実行するために，switch_to_kernel_mode と switch_to_user_mode という関数を用意した．これにより，カーネルウェア内から明示的にモード遷移を制御できる．

4.2 システムコールの関数呼び出し

AP に変更を加えなくても I/O 処理オーバヘッドを削減するため，本システムでは関数呼び出し型のシステムコールを持つ libc ライブラリを作成した．カーネルウェアは，このライブラリをリンクすることで，呼び出しオーバヘッドの少ない関数呼び出し型のシステムコールを利用できる．動的リンクを使用している AP であれば，再コンパイルの必要なくこのライブラリを使用できる．

このライブラリは，ユーザモードでも動作できるように，動作モードに応じてソフトウェア割込みか関数呼び出しかを切り替えるハイブリッド仕様になっている．大部分のシステムコールインタフェースのソースはマクロをもとに生成されているので，この変更はマクロを書き換えることで容易に実現できた．

4.3 カーネル内シンボルの解決

カーネルウェアからカーネル内資源を利用した改良を加えるためには，カーネル内シンボルの参照を解決する必要がある．カーネル内のシンボル解決には，カーネルのシンボルとアドレスとが記述されているリンカスクリプトを用いる静的な方法を採用した．カーネル内のシンボルを使用するカーネルウェアはコンパイル時にこのリンカスクリプトを使用することで，シンボルのアドレス解決を行う．

現状では静的なシンボル解決をしているために，カーネルの変更ごとにカーネルウェアの再コンパイルが必要となっている．そのため，動的リンクを使った実行時のシンボル解決が今後の課題となっている．

4.4 プリエンプション

ユーザモードで動作する AP と同様，カーネルウェアもプリエンプティブでなければならない．NetBSD はユーザモードでの割込みに対し，必要であれば他の AP へ処理を渡すことでプリエンプションを実現している．しかし，カーネルウェアはカーネルモードで動作しているため，割込みが発生してもこのままではプリエンプションが発生しない．そこで，本システムでは割込みをフックし，カーネルモードでの割込みでも，カーネルウェアに対してのものであればプリエンプションを発生させる．

この判断は，モード遷移が起こらない場合，スタックの切替えが起こらないことを利用した．もし，割込みが発生しカーネルに処理が渡ったとき，スタックポインタがユーザのメモリ空間を指していればカーネルウェアの実行中であると判断できる．なぜなら，他の AP やカーネル処理中の割込みはすべてカーネル空間のスタックを使っているためである．

割込みのフックは，LKM により IDT（Interrupt Descriptor Table）を上書きすることで実現している．これにより，割込みが発生すると，本システムが提供するフックコードが実行されるようになる．本システムを取り外す場合は元の IDT に復旧することで，OS に対する影響をなくしている．一般に，他のアーキテクチャのマシンでも，この方法は適用可能である．

4.5 安 全 性

本システムでは運用実績が豊富な AP を使用することで，安全性の向上を図る．これは，運用実績が豊富な AP であれば，異常動作を起こす確率やセキュリティホールの存在する確率が低いと考えられるためである．また，カーネルモードで実行する部分を限定することで，異常動作により OS に影響がでることを抑えることができる．

表 1 測定環境
Table 1 An environment of measurement.

OS	NetBSD–1.5.3
CPU	Celeron 500 MHz
Mem	64 MB

カーネルウェアに対して加える変更に対しては，カーネル拡張コードが及ぼす悪影響を検知する鈴鹿らの技術[3] を利用し，安全性の向上を図ることを予定している．

5. 実　　験

本システムを評価するため，3つの実験を行った．まず，本システム導入によるオーバヘッドの測定を行った．これにより本システムの導入によって，他の APへどの程度の負担がかかるか測定できる．次にシステムコールを関数呼び出し化したことでどの程度オーバヘッドを削減できるかを測定した．最後にシステムの利用例として AP のカーネルウェア化を行い，どの程度の効率化ができるか測定した．

これらの実験の測定環境を**表 1** に示す．測定時間は CPU の TSC（Time Stamp Counter）を使ってクロック単位で測定した．

5.1 システム導入によるオーバヘッド

本システムでは，プリエンプション実現のため割込みをフックしている．ここではそのフックがどの程度他の AP へ影響を与えるかを測定した．

測定方法として，割込み内での処理量が少ない getpid システムコールの処理時間を測定する．システム導入前と後を比較することで，フックが及ぼすオーバヘッドの測定ができる．ここで測定した getpid は，どちらも本システムを使わず，ユーザモードから呼び出したものである．

各 1,000 回ずつ呼び出し測定した．**図 4** は 1 回あたりのクロック数をグラフ化したものである．どちらも平均は 450 クロック程度で違いは見られない．これは割込み処理に対してフックによる影響の小さいことを示す．つまり，システム導入によって OS や他の AP の割込み処理が遅くなるような影響は出ないと考えてよい．

5.2 システムコールオーバヘッドの削減

カーネルウェアは，割込みを使わない軽量な関数呼び出し型のシステムコールを利用できる．そこで，通常のシステムコールと関数呼び出し型のシステムコールの処理時間の比較を行った．ファイル操作を行うシステムコールは MFS（Memory based File System）上で測定した．これは，ディスク I/O 処理時間のばらつきを抑えるためである．

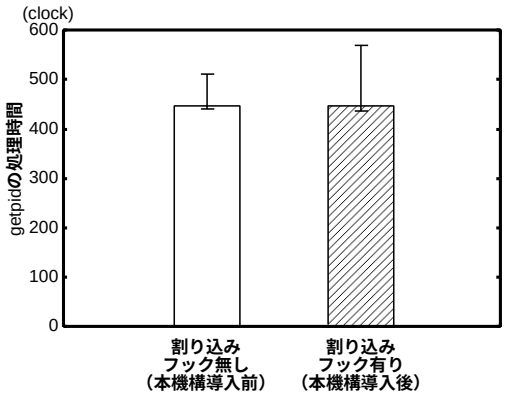

図 4 割込みをフックしたことによるオーバヘッド
Fig. 4 An overhead of interrupt hook.

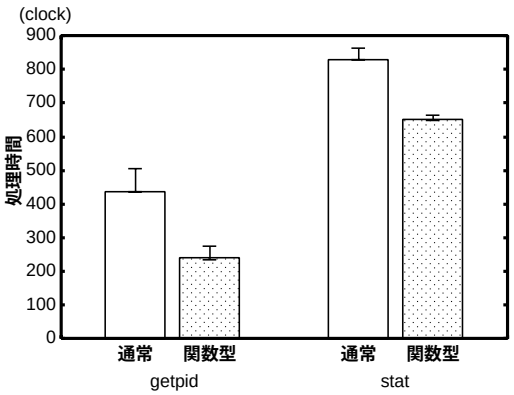

図 5 内部処理が少ないシステムコールの処理時間
Fig. 5 Processing time with a few system calls.

図 5 は，getpid や stat といった，カーネル内での処理内容が少ないシステムコールを測定したものである．getpid は約 50%，stat は約 20%程度処理時間が削減されている．

図 6 は，read と write で 64 KB のファイルを扱うといった，カーネル内での処理内容が多いシステムコールの測定結果である．この場合，両者には差が見られない．これは，呼び出しオーバヘッドの削減のみを行っているため，カーネル内の処理内容の増加とともに実行時間の比率が小さくなっているためである．

システムコールを関数呼び出し型にすることは，処理の少ないシステムコールを頻繁に呼び出す AP をカーネルウェア化する場合に有効である．また，カーネル内の処理内容が多いシステムコールのオーバヘッドを削減するには，カーネル内で動作することを利用をしてシステムコールの内部処理の改良が必要である．

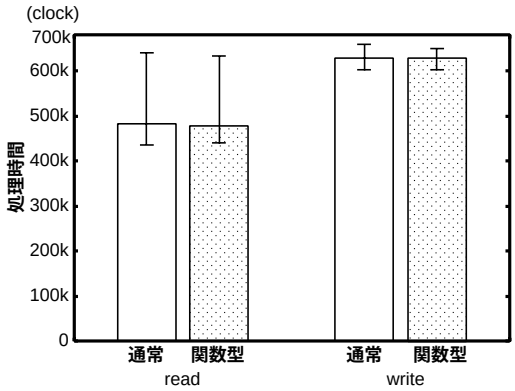

図 6 内部処理が多いシステムコールの処理時間
Fig. 6 Processing time with a lot of system calls.

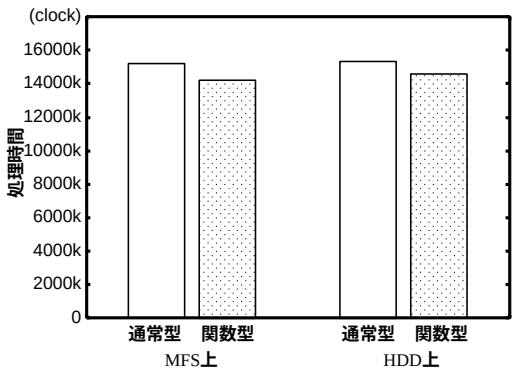

図 7 ls コマンドの処理時間
Fig. 7 Processing time of ls command.

5.3 AP のカーネルウェア化

本システムの利用例として，ls コマンドと cp コマンドのカーネルウェア化を行った．ls コマンドは stat などの処理量が少ないシステムコールを多量に使用する．そのため，ソースの変更を行わずに，システムコールを関数呼び出し型に変更するだけで，性能向上が期待できる．一方，cp コマンドは read や write システムコールの処理量が多いため，システムコールを関数呼び出し型に変更するだけでは性能向上に不十分である．そこで，カーネル内資源の直接操作を行う改良を加え，測定を行った．

5.3.1 ls コマンドのカーネルウェア化

MFS と HDD 上の 1,000 個のファイルに対し通常の ls と関数呼び出し型の ls の 2 つを使い測定を行った．本システムで実行される関数呼び出し型の ls のソースは通常の ls と同じものであり，変更は加えられていない．**図 7** が測定結果である．

関数呼び出し型の ls の方が，MFS，HDD のどちらのファイルに対しても，約 5%の処理時間が削減されている．これは，ls が stat などの処理量が少ないシステムコールを多量に使用しており，システムコール呼び出しオーバヘッドの削減が性能向上に結び付いているためである．

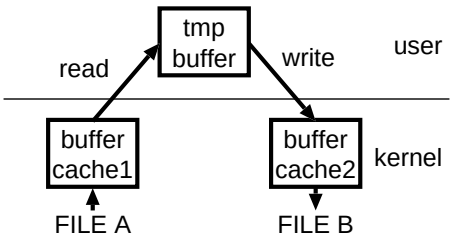

図 8 通常のファイルコピー
Fig. 8 An ordinary file copy.

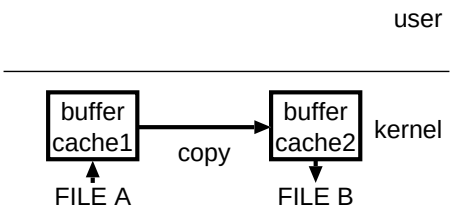

図 9 バッファキャッシュ間のファイルコピー
Fig. 9 A file copy between buffer caches directly.

5.3.2 cp コマンドのカーネルウェア化

通常の cp は**図 8** のようにユーザ空間のバッファを介してファイル内容をコピーする．この際，ユーザ空間のバッファに変更は加えられないため，このコピーはカーネル内であれば無駄である．そこで，カーネル内資源を直接操作して**図 9** のようにユーザ空間のバッファを介さずに，バッファキャッシュ間で直接コピーするカーネルウェアを作成した．

このカーネルウェアは，通常の cp コマンドのソースにバッファキャッシュ間コピー用関数を呼び出すためコードを追加し作成した．追加したコードの量は，通常の cp のソース 897 行に対し 11 行だけであり，全体の約 1.2%と非常に少なくすんでいる．追加した 11 行のコードには，ユーザモードで呼ばれた場合に read と write システムコールを使った通常の cp 処理にジャンプしたり，エラー処理するためのコードも含まれている．

MFS 上の測定結果を**図 10** に示す．測定は通常の cp，関数呼び出し型システムコールを使った cp，そしてバッファキャッシュ間コピーを行う cp を使用した．通常の cp 以外は，本システムを使用している．関数呼び出し型システムコールを使った場合，性能向上はみられなかった．これは，図 6 で示したように，read

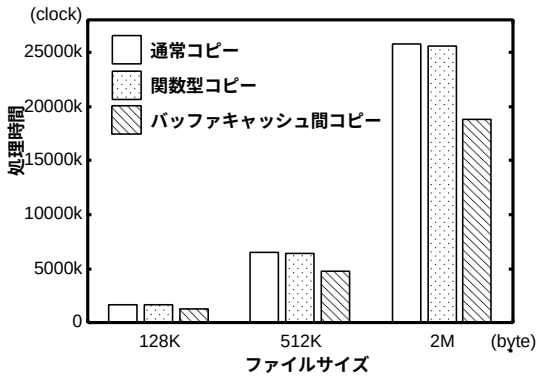

図 10 MFS 上のファイルコピー時間
Fig. 10 Processing time of file copy on MFS.

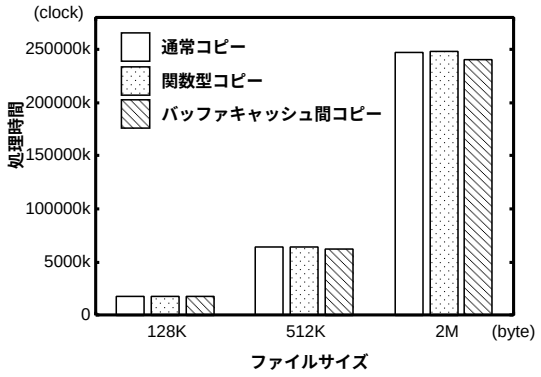

図 11 HDD 上のファイルコピー時間
Fig. 11 Processing time of file copy on HDD.

と write システムコールの処理が大きいためである．一方，バッファキャッシュ間のコピーを行った場合は，2 MB のファイルコピーの際，約 26%の処理時間の削減が実現され，カーネル内資源の直接操作の有効性が確認できた．

HDD 上のファイルに対し測定を行った結果を**図 11** に示す．MFS に比べ HDD へのアクセスが発生するため，処理時間削減の割合が最大で約 3%に低下している．より処理時間の削減を図るには，田胡ら[4] が述べているような，データコピーに CPU が介在しないゼロコピー技術を利用することが考えられる．

6. 関 連 研 究

ここでは，関連研究について述べ，本機構との比較を行う．

6.1 カーネルモジュール

カーネルモジュールとして AP のサービスを実現する手法である．カーネル内で動作するため，モード遷移のオーバヘッドが必要なく，さらに，カーネル内資源を直接操作して処理内容に特化した改良を加えることが可能である．

たとえば Linux の kHTTPd[2] はカーネル内で Web サービスを行うカーネルモジュールである．kHTTPd はモード遷移を必要としないことに加え，カーネル内の低レベル操作を使いデータコピーの少ないファイル転送を実装しているため，ユーザランドで動作する Web サーバよりも高い性能を実現している．

本システムと比較して，カーネルモジュールはゼロから作成するため，OS よりに実装され，高い性能を発揮することが予想される．しかし，本システムでは既存 AP を利用できるため，開発効率の面で有利である．

6.2 特殊システムコール

ファイル転送を効率的に行う sendfile のように，ある特定の処理に特化したシステムコールを AP に提供する手法である．

カーネルウェアでは，システムコールを使う以外に，カーネル内資源を直接操作する改良を処理内容に応じて柔軟に加えることができる．

6.3 専 用 OS

Exokernel[5] は計算機資源管理ポリシを AP 側に任せることが可能な OS である．AP 側で計算機資源管理を行うことで，オーバヘッドが少ない計算機資源利用が可能である．

SPIN[6] は AP の一部のコードをカーネル内に取り込み，実行することができる OS である．カーネル内でコードを実行することでモード遷移を抑え，オーバヘッドの少ない資源利用を行える．カーネル内実行コード記述に Modula-3 を使うことで安全性を高めている．

専用 OS を用いて効率的な I/O 操作を AP に提供する手法に対し，本システムは汎用 OS 上に実装した．汎用 OS を使用することで，動作実績のある AP を豊富に利用できる．

6.4 カーネル内実行 AP

Kernel Mode Linux[7] は AP をカーネル内実行する機構である．実行コードは型検査されるため，安全性を確保することができる．

Cosy[8] も Kernel Mode Linux と同様，AP をカーネル内実行する機構である．実行時にコードの安全性を検査する．また，システムコールの内部処理を改良することで，コピー回数の削減などを実現している．

本システムでは，カーネル内資源を直接操作できる枠組みを用意することで，カーネルウェアの処理内容に最適化した I/O 処理手段を提供している点が異なる．

7. まとめと今後の予定

本論文では AP を変更することなく汎用 OS のカーネル内で実行し，カーネルウェアとして OS 機能を拡張する手法を提案し，実装した．カーネル内で実行することにより，カーネルウェアは I/O 処理を行う場合，モード遷移のオーバヘッドを受けずにすむ．さらに，カーネル内資源を直接操作する枠組みを提供することで，開発者はカーネルウェアの I/O 処理内容に特化した改良を加えることができる．また，すでに存在する豊富な AP をカーネルウェア開発の土台として利用できる．

実験により，本システム導入による OS や他の AP へ与えるオーバヘッドは少ないこと，システムコールを関数呼び出しにすることで，モード遷移のオーバヘッドを削減できることを確認した．また，システム利用例として，cp コマンドのカーネルウェア化を行い，カーネル資源の直接操作を行う改良を加え性能が向上したことを確認した．

今後の予定として，Web サーバのような大きな AP をカーネルウェア化し，I/O 処理オーバヘッドの削減方法について考察し，カーネルウェアで共有できる改良方法をライブラリ化する．また，安全性の向上や，カーネル内シンボルの動的解決について検討する．

参 考 文 献

1) Joubert, P., King, R.B., Neves, R., Russinovich, M. and Tracey, J.M.: High-Performance Memory-Based Web Servers: Kernel and User-Space Performance, *Proc. USENIX Annual Technical Conference* (2001).
2) van de Ven, A.: kHTTPd Linux HTTP Accelerator Web Pages. http://www.fenrus.demon.nl/
3) 鈴鹿倫之，中村嘉志，多田好克：カーネル拡張のための効率的な開発環境，情報処理学会第 41 回プログラミング・シンポジウム報告集，pp.57–64 (2000).
4) 田胡和哉，根岸康，奥山健一，村田浩樹，松永拓也：オープンソフトウェアによる Network Attached Storage の性能の解析および改善に関する一試み，情報処理学会論文誌，Vol.44, No.2, pp.344–352 (2003).
5) Engler, D.R., Kaashoek, M.F. and James O'Toole Jr: Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp.251–266 (1995).
6) Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M.E., Becker, D., Chambers, C. and Eggers, S.: Extensibility, Safety, and Performance in the SPIN Operating System, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp.267–284 (1995).
7) 前田俊行，住井英二郎，米澤明憲：Linux/TAL: 型付きアセンブリプログラムのカーネルモード実行方式，日本ソフトウェア科学会第 4 回プログラミングおよびプログラミング言語ワークショップ論文集 (2002).
8) Purohit, A., Wright, C.P., Spadavecchia, J. and Zadok, E.: Cosy: Develop in User-Land, Run in Kernel-Mode, *Proc. 9th Workshop on Hot Topics in Operating Systems* (2003).

(平成 16 年 1 月 31 日受付)
(平成 16 年 5 月 9 日採録)

佐藤 喬（正会員）

2002 年電気通信大学大学院情報システム学研究科博士前期課程修了．2004 年同博士後期課程退学．現在同研究科助手．システムソフトウェアの研究に従事し，現在組み込みシステムに興味を持つ．

安田 絹子（正会員）

1998 年慶應義塾大学大学院政策・メディア研究科博士課程前期修了．2001 年同大学院政策・メディア研究科博士課程後期満了．2002 年電気通信大学大学院情報システム学研究科助手．2004 年 4 月より慶應義塾大学 SFC 研究所訪問研究員およびインテリシンク（株）エンジニア．モバイルコンピューティング，システムソフトウェア，分散ファイルシステム等に興味を持つ．

中村 嘉志（正会員）

1994 年神奈川大学理学部情報科学科卒業．1996 年電気通信大学大学院情報システム学研究科博士前期課程修了．1997 年同専攻博士後期課程退学．同年同研究科助手を経て，現在産業技術総合研究所特別研究員．分散システムの研究に従事し，現在情報支援システムに興味を持つ．IEEE，電子情報通信学会各会員．

多田　好克（正会員）

1985 年東京大学大学院工学系研究科情報工学専門課程博士課程修了．工学博士．同年電気通信大学電子情報学科着任．1992 年より電気通信大学大学院情報システム学研究科．並列・分散システムの記述法に興味を持ち，オペレーティングシステムをはじめとするシステムソフトウェアの実現法に関する研究に従事．ACM，電子情報通信学会会員．