自動モノリシック化機能を持つ組込み向けマイクロカーネル構成オペレーティングシステム

久住憲嗣[†1] 北須賀輝明[†2]
中西恒夫[†2,†3,†4] 福田晃[†2]

組込みシステムが大規模化している現在，組込み向けオペレーティングシステム（OS）の保守性や開発効率が重要である．そこで，本研究ではマイクロカーネル構成を採用した組込みシステム向けOS Lambdaの設計・実装を行っている．マイクロカーネル構成はモノリシックカーネルに比べ開発効率は良いが，性能が悪く，メモリを余分に消費する傾向にある．この問題を解決するために，組込みシステムでは出荷後にソフトウェアを変更することはまれである性質を利用し，実装時にはマイクロカーネル構成で実装を行い，出荷時にはモノリシックカーネルに自動変換を行い，性能の改善を図る手法を提案する．この手法を用い，マイクロカーネルを設計・実装し性能測定を行った結果，本手法が有効であることを確認できた．

Design and Implementation of the Lambda μ-Kernel Based Operating System for Embedded Systems with Automatic Transformer to Monolithic Structure

Kenji Hisazumi,[†1] Teruaki Kitasuka,[†2] Tsuneo Nakanishi[†2,†3,†4] and Akira Fukuda[†2]

With large-scale of embedded systems, improvement of development efficiency is one of the most important problems. In this paper, we design and implement an embedded operating system, called the Lambda operating system, which improves the maintainability and development efficiency of the operating system. The Lambda operating system employs micro-kernel architecture, which allows the operating system to be easily designed. In addition, we propose a method to improve operating system performance by reconstructing it in implementation. With the method, Lambda is implemented as monolithic kernel. The method allows the operating system to be quickly developed and gives high performance. This paper also shows that the method is useful through implementing a prototype of Lambda and its performance evaluation.

1. はじめに

近年，情報家電の登場などにより，組込みシステムが大規模化してきている．大規模化するにつれ開発工数が増加する反面，競争の激化などにより製品を短期間に設計・開発し，市場に出すことが求められている．このため組込みシステムの開発において，開発工数の削減が必要不可欠であり，開発効率の向上が重要な課題となっている．組込みシステムの開発効率を向上させるために，組込み向けオペレーティングシステム（OS）の保守性や開発効率が重要であるにもかかわらず，この点に注目しOSを実装している例は少ない．そこで，本研究では，OSの保守性や開発効率を向上させるため，マイクロカーネル構成を採用した組込み向けマイクロカーネルOS Lambdaの設計，実装を行う．

しかしながら，マイクロカーネル構成ではプロセス間通信を使用することにより，従来のモノリシック構成のカーネルよりも性能が悪いという欠点を持つ．これは，L4[1)]などにより大幅に改善されてはいるが，依然モノリシック構成のほうが性能が良い．さらに，各

†1 九州大学システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†2 九州大学システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University
†3 九州大学システムLSI研究センター
System LSI Research Center, Kyushu University
†4 九州大学情報基盤センター
Computing and Communications Center, Kyushu University

機能をプロセスやスレッドに分割することにより，各プロセス・スレッドごとにスタック，プロセス管理ブロックなどが必要なので，メモリを多く消費する傾向にある．これらの欠点は，汎用 OS であれば問題にならないことも多いが，組込み用途のような資源を大幅に制限されたシステムでは致命的な欠陥になりうる．

そこで本稿ではさらにマイクロカーネル構成の各機能の独立性の良さを生かしつつ，組込みに特化し，性能やメモリ消費量を改善する手法の提案を行う．

2. 組込みシステムにおける保護

2.1 保護の重要性

システムの開発効率やセキュリティ，信頼性の向上のために，組込みシステムでも，現在保護の必要性が叫ばれている．本章では組込みシステムが必要とする保護の性質について整理する[2)]．

まず，保護の対象となるものを列挙する．

- メモリ/アドレス空間
  メモリやアドレス空間を分離，保護することによって，関係の薄いプロセスの影響を受けないようにする．
- プロセッサ時間
  1 つの実行実体がすべてのプロセッサ時間を費やしてしまうことによる，他の実行実体への悪影響を防ぐ．
- 各種資源
  アクセスするはずのないプロセスが資源をアクセスし，資源を勝手に使用することを防ぐ．

保護機構の用途には以下のようなものがある．

- カーネルテキストやデータの破壊防止
  実行時にカーネルを破壊し致命的な状況に陥らないようにするための保護で，保護の用途としては古典的なものである．メモリ保護を使用する．
- セキュリティ保護
  組込みシステムでもソフトウェアをダウンロードし，実行する応用が増えてきており，そのようなソフトウェアが，カーネルや他のソフトウェアの動作に支障をきたさないように，保護する必要がある．メモリ，プロセッサ時間，資源を的確に保護しなければならない．
- 他社ソフトウェアなどの信頼できないソフトウェアの隔離
  組込みシステムでも，WWW ブラウザなどの他社製ソフトウェアを搭載する機会が増えてきているが，そのソフトウェアがシステムへ悪影響を及ぼさないようにするために保護が必要となる．メモリ，プロセッサ時間，資源の保護が必要である．
- デバッグ支援のための保護
  不正なメモリアクセスや各種資源へのアクセスが，潜在的なバグとして残りやすい．しかし，使用しないはずのメモリ空間や資源へのアクセスをあらかじめ制限しておくことによって不正なアクセスを検出し，デバッグ支援に利用できる．この場合は，保護機構は完全である必要はなく，また，組込みでは開発終了後は保護を無効にできることが望ましい．

携帯電話などにおいて，ダウンロードしてきたソフトウェアを実行する機会が増えてきた現在，保護機構を持たないプロセッサや OS を採用してきた組込みシステムの分野においても保護機構が必要であると考えられる．マイクロカーネル構成の OS は OS の機能ごとに保護を設定することができ，また，柔軟に構成することが可能であるので，保護に適した OS の構成方法である．

2.2 保護が不必要な場合

組込みシステム上で動作するソフトウェアには，動的にアプリケーションを動作させる必要のない，固定的な部分が存在する．そのような部分で，特に保護を必要としない部分においても，デバッグ支援としての保護は有効である．しかし，開発が終了し安定した後には，保護を無効にし，性能の向上や資源の節約につとめた方がよい．

マイクロカーネルは，各機能を，アドレス空間の分離による保護で分断して，独立性を向上させた結果，機能呼び出しをプロセス間通信（IPC）に頼らざるをえない．IPC の速度向上は L4[1)] で図られており，汎用システムの分野では問題となりにくいレベルにある．組込みシステムの固定的なソフトウェアに関しては，さらに保護を無効にして，性能の向上を図ることができれば，省資源，省電力につながり，組込みシステムにおいては都合がよい．

そのようなの手法の 1 つとして，文献 3) では，サーバをカーネル空間で動作させることによって，アドレス空間の切替えオーバヘッドを削減し大幅な性能向上を図っている．しかし，同じ空間で動作しているにもかかわらず，カーネルの機能を利用しプロセス間通信しなければならず，依然オーバヘッドが残る．

本研究では，機能呼び出しのオーバヘッドをさらに削減するためにプロセス間通信を関数呼び出しに静的に置換する手法の提案を行う．

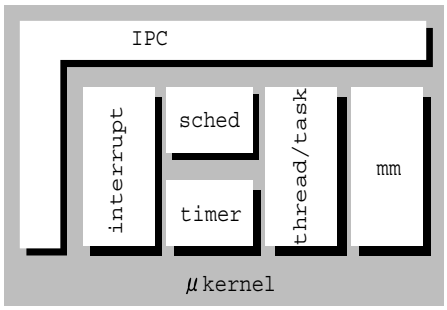

図 1 μ-kernel の構成
Fig. 1 Structure of the μ-kernel.

3. 組込み向けマイクロカーネル OS Lambda

Lambda は本研究室において開発を行っている，組込みシステム向け OS である．Lambda は現在多く使用されている組込み向け OS とは違い，保護を重要視した OS である．また，デバッグ支援のための保護を特に重要視しており，保護が必要がない場合には保護をとりはらうことができるように設計が行われている．Lambda のマイクロカーネルは以下のモジュールから構成される（**図 1**）．

- スレッド・プロセス管理機構（図 thread/task）
- 割込み管理機構（図 interrupt）
- タイマドライバ（図 timer）
- メモリ管理機構（図 mm）
- プロセス間通信（図 IPC）
- スケジューラ（図 sched）

また，Lambda では以下のプロセッサや環境のサポートを行う．

- SH-3[4)]
- IA32[5)]
- Linux

これらのアーキテクチャ以外でも，ページング機構を持つプロセッサであれば，移植は容易である．

3.1 プロセス間通信

Lambda に IPC を実装するにあたり，高速であること，およびデータの受け渡しが柔軟にできることを目標とした．そこで，遠隔手続き呼び出し（RPC）を使用することを前提とした IPC プリミティブの提供を行い，また，メモリのコピー，マッピングを柔軟に行うことができるようなデータ転送モードの実装を行う．

IPC プリミティブとしては以下のようなものの実装を行う．

- call
  遠隔手続き呼び出しを行うために使用するモードである．受信側スレッドに引数を渡し，返答がくるのを待つ．
  リクエストの送信のために送信デスクリプタと送信引数を，返り値の受信のために送信スレッドと受信ディスクリプタ，受信引数を指定する．返答を受信するか，タイムアウトするまでブロックされる．
- send_reply_and_recv
  RPC などのサーバの性能を向上させるために，値を返答する動作と，引数を受信する動作を 1 つにまとめたモードである．1 つにまとめることにより，通常だと send，recv と 2 回システムコールを発行しなくてはならないところを，一度のシステムコール呼び出しで処理が完了し，コンテキストスイッチ回数が減り性能が向上する．返答のために送信デスクリプタと送信引数を，リクエストを受信するために送信側スレッドと受信ディスクリプタ，受信引数を指定する．
  受信が完了するか，タイムアウトされるまでブロックされる．
- send
  送信のみを行うモードである．
  送信デスクリプタ，送信内容を指定する．
  受信側により受信されるか，またはタイムアウトするまでブロックされる．
- recv
  受信のみを行うモードである．
  送信側スレッドと受信デスクリプタ，受信内容を格納する変数を指定する．
  送信側がメッセージを送信し受信が完了するか，またはタイムアウトするまでブロックされる．

また，データ転送モードには以下のようなモードがある．

- copy
  送信元から送信先へデータのコピーを行う．
- map read/write
  領域を共有メモリとして利用するために，送信元のページを送信先のページに書き込み許可の状態でマップする．送信元はページ境界に注意して転送を行わないと，送信先にデータを破壊されてしまう可能性がある．
- map read only
  送信元のページを送信先のページに read only の状態でマップする．書き込まれたくない場合や，読み込みのみで使用する場合にこのモードで転送する．送信元はページ境界に注意して転送を行わないと，送信先に望まないデータを読み込まれて

しまう可能性がある．

すべてのデータ転送モードを copy on write として実装してもよいのだが，機構が複雑であり，またコピーする必要のない用途で使用することも多いので，細かくデータ転送モードを指定できる方法を採用する．

3.2 メモリ管理

カーネルとは別にメモリ管理サーバを実装し，物理ページの管理や割当てを行わせる．カーネルやユーザプロセスは，必要に応じて IPC を用いてメモリ管理サーバにページ割当て要求を出し，ページを割り当ててもらう．ページは 3.1 節で述べた IPC のマッピングの機構を用い，必要なアドレスにマップを行う．メモリ管理サーバは，開発時にはプロセスとして実行されるが，開発終了後はカーネルスレッドとして実行し，後述のモノリシック化手法を用いてカーネル空間におけるメモリ割当ての高速化を図る．

4. プロセス構造の変更

4.1 プロセス構造を変更することによる IPC 動作の違い

実装時には各機能を通常プロセスとして実装・デバッグを行い，出荷時にはカーネル内スレッドとして実行することにより，開発効率と性能の両立を図る．つまり，スレッドが所属するプロセスが開発途中に変化する可能性がある．この手法を実現するために，Lambda はプロセス構造の変化に対応しなければならない．プロセス構造が変化したときに，IPC の動作，特にデータ転送方法をどのように変化させる必要があるかを考察し，実装の説明を行う．

- copy
  - 同プロセス
    同じプロセス中のスレッド間でデータのコピーを行う場合には，アドレス空間が同じであるので，直接データのコピーを行う．
  - 別プロセス
    別プロセスに属するスレッド間でデータのコピーを行う場合，アドレス空間が違うので直接データをコピーすることはできない．そこで，送信元のデータを，受信先のカーネル空間にマップし，受信先においてマップした領域から，受信先のユーザ空間にデータのコピーを行う（**図 2**）．
- map read only
  送信側スレッドが指定したページを，受信側スレッドが指定した領域に書き込み禁止属性でマップする．送受信スレッドが同プロセスに所属していても，別プロセスでも，書き込み禁止属性で受信側が指定した領域にマップを行うこととする．同プロセスであればマップが必要ないように思えるが，書き込み禁止属性を指定するためにマップを行う必要がある．
- map read/write
  送信側スレッドが指定したページを，受信側スレッドが指定した領域に書き込み許可属性でマップする．送受信スレッドが同プロセスに所属していても，別プロセスでも，書き込み許可属性で受信側が指定した領域にマップを行うこととする．

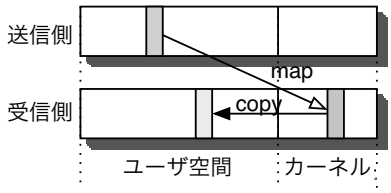

図 2 プロセス間コピー
Fig. 2 Interprocess copy.

4.2 ユーザ空間でのマップ空間の調停

本システムではスレッドがどのプロセスに所属するかを変更することを前提としている．スレッドのプログラムテキストやデータは，リロケータブルな実行形式を用いることにより，所属を変更することが可能になる．

しかし，Lambda 上のソフトウェアでは，カーネルに領域を指定してメモリをマップしてもらうという操作をよく行う．これは，マップをともなう IPC を利用するときに使用される．このとき，プロセス構造が変化しないのであれば，ある程度静的にアドレス割当て問題を解決することができるが，あらかじめ所属するプロセスが分からない Lambda では，破綻をきたす可能性が高い（**図 3**）．そこで，Lambda の標準ライブラリとして，プロセスごとにアドレス空間の調停を行う標準的な手続きを提供する．この手続きは各プロセスにそれぞれリンクしておく．そして，この手続きはリンクされたプロセス内のアドレス空間をユーザに割り当てるように実装を行う．これにより，プロセスごとにアドレス空間を管理することができ，この手続きを使用している限りは，プロセスが統合された場合でもアドレス空間の衝突などを起こすことがなくなる．

たとえば，あるソフトウェアがユーザプロセス (a) から別のプロセス (b) にリンクされた場合を考える．(a) のプロセスにリンクされているときは，(a) 内の手続きによりアドレス空間の割当てを受ける．この状態から (b) にリンクされるように変更した場合，(b)

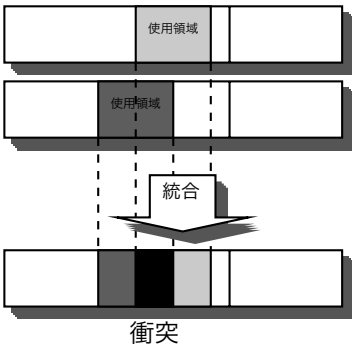

図 3 使用アドレス空間の衝突
Fig. 3 Collision of used address spaces.

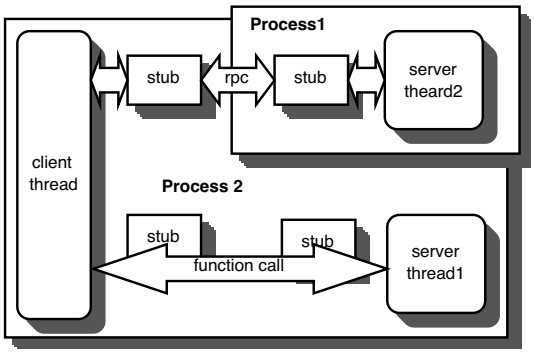

図 4 サーバスタブとクライアントスタブ
Fig. 4 Server stub and client stub.

のプロセス内の手続きによりアドレス空間の割当てを受けるようになり，(b) の他のソフトウェアが使用しているアドレス空間と衝突することがなくなる．

5. インタフェースジェネレータ

5.1 LIG

RPC を簡便に利用可能にし，また，モノリシック化の実現のために，Lambda の API に対応したインタフェースジェネレータ「LIG」の実装を行う．LIG は通常のインタフェースジェネレータのように，インタフェース定義からクライアント側・サーバ側のソーステンプレートおよびスタブの生成を行う．

5.2 モノリシック化

LIG ではモノリシック化を支援するための機能の実装を行う．モノリシック化とは，同プロセスに配置したスレッド間での不必要な IPC を関数呼び出しに置き換えることである．すべてのスレッドを 1 プロセスに配置し，IPC を関数呼び出しに置き換えられたシステムは，すでにマイクロカーネル構成ではく，モノリシックカーネルなので，この手法をモノリシック化と呼ぶ．

RPC を使用する際には，スタブと呼ばれる小さなコードを使用する．クライアント側スタブはクライアントから呼ばれた際に，引数をコード化し，サーバにリクエストを伝える役目を果たす．そして，サーバ側のスタブがそのリクエストを受け，引数を復号し，サーバに渡して処理を行う．

このスタブの生成を工夫し，クライアントとサーバが同じプロセス内で動作している場合には関数呼び出しを行い，違うプロセスの場合には通常どおり RPC を行うように実装を行う（**図 4**）．この手法を使用することにより，RPC を関数呼び出しに置き換えることができる．

5.3 モノリシック化の実装

本節では RPC のモデルを以下のようなものと考え，モノリシック化の具体的な手法の議論を行う．RPC クライアントは RPC サーバスレッドに IPC の call モードを利用し，リクエスト行う．RPC サーバとしては通常ではスレッドが 1 つ起動されており，そのサーバがリクエストを受信し，サーバ側の処理を行い，クライアントに返答を行う．その際には，send_reply_and_recv モードが使用される．実装時の設定により，RPC リクエストごとにスレッドを起動するモードや，あらかじめ複数のサーバスレッドを起動しておき，サービスを行うモードを選択できる．

このようなモデルにおいて，RPC を関数呼び出しに置換する，モノリシック化の実装について説明を行う．

(1) 下のような文法を持つインタフェース記述言語で，提供したいサービスのインタフェースを定義する．

```
int rpc_call(out int arg1,
             in  int arg2);
```

(2) LIG は，そのインタフェース定義を元に下記のようなクライアントが使用するためのプロトタイプ宣言を生成する．

```
int rpc_call(int arg1,
             int *arg2);
```

(3) クライアント実装者は，このプロトタイプ宣言を元にクライアントの実装を行う．

(4) また，LIG は下記のようなサーバが実装すべき手続きのプロトタイプ宣言を生成する．

```
int rpc_call_impl(int arg1,
                  int *arg2);
```

(5) サーバ実装者はこのプロトタイプ宣言を元に，サーバ手続きの実装を行う．

(6) LIG はクライアントのコンパイル時にクライア

ント用のスタブを生成する．プロセス構造定義により，RPC を行うスタブを生成するか，手続き呼び出しを行うスタブを生成するかを指定する．

```
/* 同じアドレス空間に配置されていて
   関数呼び出しに置き換える場合 */
int rpc_call(int arg1,
             int *arg2)
{
   return rpc_call_stub (....);
}

/* 違うアドレス空間に配置されていて
   RPC を用いる場合 */
int rpc_call(int arg1, int *arg2)
{
   int retval;
   /* 引数コーディング */
   call (....);
   /* 返値のデコーディング */
   return retval;
}
```

(7) LIG はサーバのコンパイル時に，RPC サーバ，手続き呼び出しスタブの両者を生成する．しかし，プロセス構造定義により，RPC サーバの削除を指定した場合は，手続き呼び出しスタブのみを生成する．

```
/* RPC サーバスタブ */
void rpc_call_server ()
{
   int arg1, arg2, retval;

   recv (....);
   for (;;)
   {
      /* 引数のデコーディング */
      retval
        = rpc_call_impl (arg1,&farg2);
     /* 返値のコーディング */
      send_reply_and_recv (....);
   }
}

/* モノリシック化用スタブ */
int rpc_call_stub (int arg1,
                   int *arg2)
{
   return rpc_call_impl (arg1, arg2);
}
```

このように，必要のない RPC を関数呼び出しに置換し，性能の向上を図る．

記述の関係上，モノリシック化した場合にもスタブ関数を呼び出し，またスタブ内でサーバ関数を呼び出すように記述しているが，実際にはインライン展開やマクロを用いて，1 度の関数呼び出しに置き換えを行う．

ユーザが実装したサービスを RPC サーバとして利用する状態では，カーネルの IPC 機構により排他制御が行われ，特に指定しない限り RPC サーバをリエントラントに実装する必要はない．ところが，LIG によってモノリシック化された場合には，IPC 機構による排他制御が行われないため，サーバ関数はリエントラントである必要がある．リエントラントなサーバ関数の実装を強制するのは実装者に負担が大きいため，関数呼び出しを実行する際に，スタブ内で排他 lock を行う．また，サーバ関数全体を lock する方法では性能が低い場合には，実装者は特にサーバ関数をリエントラントに実装し，その事実を LIG に伝えることによって，排他 lock を行わないように実装を行う．リエントラントであれば複数のコンテキストによりサーバ関数を実行することが可能になり，通常の関数呼び出しとほぼ同等の性能が期待できる．

5.4 モノリシック化にともなう問題点

本節ではモノリシック化を行うことによって生じる各種制限事項について説明する．

前述した RPC のモデル以外では正しく動作しない可能性がある．例として，サーバ側スタブがリクエストの受信時にサーバスレッドを生成し，処理を行わせる RPC モデルについて考える．サーバ側の処理が終了したのちに返答を返す場合，つまり同期して実行されることを期待する場合には，サーバがリエントランに実装されていれば，特に問題はない．ところが，サーバが処理を終了する前に返答を返し，その後にサーバが処理を行うような，非同期に実行されることを期待する場面においてモノリシック化を行うと，すべての処理が終了するまでクライアント側がブロックされてしまい問題が発生する．

現在の RPC の実装では，RPC サーバは関数の形で実装し，1 つのリクエストごとに要求を受け取り，返答を返す形になっている．この形では，初期化・終了処理を簡単に行えず，不便である．今後，初期化・終了関数を登録しサーバ起動時に呼び出されるように実

装を行う．

RPC のモノリシック化を行うと，サーバ側関数はクライアントのコンテキストで動作することになる．クライアントのコンテキストで動作すると，スレッドの情報を取得して動作の変更を行うもの，たとえばスレッド id を利用して同期を行う場合に問題となる．この問題を根本的に解決するのは困難であるが，サービスごとに同期機構の抽象化を行い，直接スレッド id を利用せずに同期を行える機構を提供することにより解決可能である．

RPC の引数を COPY モードで渡す関数のモノリシック化を単純に行うと，引数のアドレスを渡すかたちになるため，サーバ側で引数を変更するとクライアント側のデータも書き換わり，互換性がなくなる．COPY 指定されたときはデータのコピーをサーバスタブ内で行うように実装する．データのコピーはオーバヘッドが大きいため，必要がない場合には MAP 指定を行う方が望ましい．

ソースコードを提供されていない場合にはモノリシック化が行うことができない可能性がある．ライブラリとして提供されている場合においては，適切なラッパを作成することにより，RPC で機能の呼び出しを行うこともでき，また，関数呼び出しで機能の呼び出しをすることが可能である．しかし，サーバ側がバイナリのみで提供され，かつ IPC 経由でのみ呼び出し可能な場合は，関数呼び出しに置換することはできない．しかし，バイナリがリロケータブルであれば，アドレス空間の統合は可能であり，その分の性能は向上する．クライアント側がバイナリのみで提供され，IPC 経由でのみ機能を呼び出すように実装されている場合でも同様である．

次に，サーバ提供者はソースコードを提供したくないが，モノリシック化は可能にしたい場合を考える．そのような場合には，RPC 用，関数呼び出し用の両方のサーバスタブをあらかじめリンクしておき，ユーザに提供を行う．また，クライアントスタブも提供する．ユーザは，サーバプログラムを同じプロセスとして実行する場合には関数呼び出し用のスタブをリンクするように設定する．また，違うプロセスとして実行する場合には RPC 用のスタブをリンクするように設定する．このようにすると，サーバ提供者がソースコードを提供しなくても，モノリシック化することが可能になる．

6. 評　　価

実装を行っている Lambda において，別プロセス間 RPC，同プロセス内 RPC，モノリシック化 RPC にかかる時間の測定を行った．測定は PC/AT 互換機 Celeron 300A 上で行った．また，時間の測定は，Pentium から IA32 に導入された rdtsc 命令を用いた．この命令は，プロセッサ内部のクロックごとにインクリメントされる，タイムスタンプカウンタを読み出す命令である．この命令を測定したい区間の始点と終点に挿入し，2 点のタイムスタンプの差を求め測定時間とした．測定内容は，RPC を使用し機能呼び出しを行い，その応答が返ってくるまでの時間である．パラメータとして渡されるデータは 0 から 300 バイトまで可変した．**図 5** のように同プロセス内 RPC (a)，プロセス間 RPC (b)，モノリシック化 RPC (c) の 3 種類のモードにおいて応答時間の測定を行った．プロセス間 RPC，同プロセス内 RPC については，データの受け渡し方法として，コピーによるもの，書き込み禁止マッピングの 2 種類を比較する．また，モノリシック化 RPC では参照渡しによりデータの受け渡しを行う．すべての場合において，RPC 要求を出した直後に対象サーバが要求を処理するようにスケジューリングする．また，モノリシック化 RPC の場合は，必ずロックが取得でき，待たされることはない．以上の結果を**図 6** に示す．また，参考のために基本的なシステムコールの応答時間を**表 1** に示す．thread_create はスレッド作成，process_create はプロセスを作成するシステムコールである．thread_sched はスレッド切替えを行うシステムコールであるが，他プロセスに切り替えるときは，アドレス空間の切替えも同時に行う．

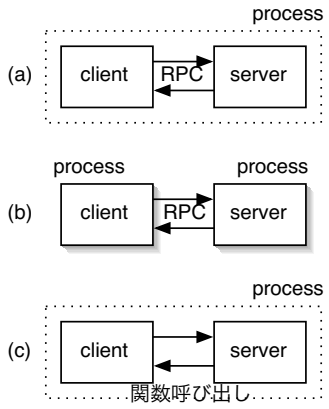

図 5 性能測定時の構成
Fig. 5 RPC structures of the performance measurement.

同プロセス間 RPC はマッピングによるデータの受け渡しの方が低速だが，これはデータの転送量が少ないために，メモリマップを変更するための処理の方が時間がかかっているためである．また，全体的に，デー

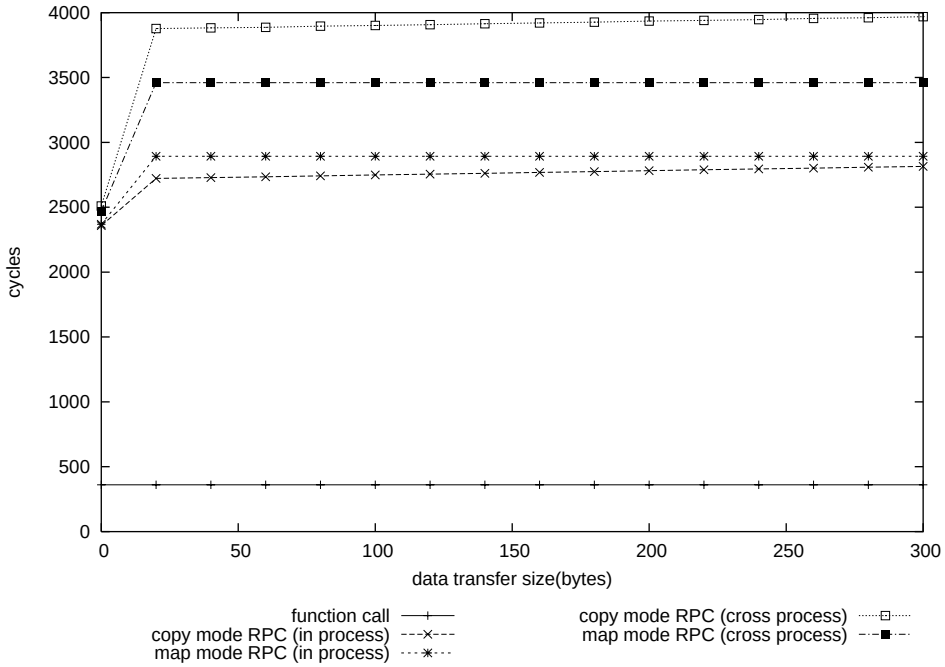

図 6 各種 RPC の性能測定結果
Fig. 6 Result of the performance measurement of various RPCs.

表 1 基本的な system call の応答時間
Table 1 Response time of basic system calls.

機能		サイクル数
thread_create		3630
process_create		30000
thread_sched	同プロセス	411
	別プロセス	1986

タサイズが 0 バイトのときが高速になっているが，これはデータがない場合にはパラメータの処理を行わない実装になっているためである．

図を見ると，プロセス構造を変化させ，サーバとクライアントを同プロセスに配置することによって，20%程度性能が向上していることが分かる．しかしながら，RPC をモノリシック化 RPC，すなわち関数呼び出しに置き換えることによって，10 倍以上の高速化が可能になる．

7. 関 連 研 究

モノリシック化のアイデアに関する研究は著者らの知る限り存在しないので，比較的モノリシック化手法に近い LRPC，および，Lambda の構成法の参考としたコンポーネントベース OS の紹介を行う．

LRPC[6] は同一マシーン上の別プロセス間での RPC を高速化する手法である．送信側スタックを受信側にマップしてパラメータを渡すことにより，データの受け渡しを高速に行う．参照渡しを行うデータに関しては，送信側スタックにデータをコピーする．これに対し本研究では，同一プロセス内での RPC を，静的に関数呼び出しに置換し高速化を図る．LRPC のようにプロセスをまたがる RPC を実行することはできないが，関数呼び出しに置換することによって，高速に RPC を実行することができ，その際のオーバヘッドは微少である．また，静的に置換することによって，コンパイラにより最適化を施すことも可能になる．このような性質は，ソフトウェアが固定的な部分を多く持つ組込みシステムに適した性質である．

最近では，組込みシステム開発のためにコンポーネントベース OS が注目されている[7)~9)]．コンポーネントベース OS は，組込みシステム上のアプリケーションがあまり変化しないことに注目し，必要な機能のみを組み合わせ，アプリケーションに特化した OS を自動的に生成することを目標としている．これらの研究では，コンポーネント分割手法やアプリケーションを調査し必要なコンポーネントを選び出す手法，コンポーネントを組み合わせるときの高速化手法などについて主に研究を行っている．しかしながら，コンポーネント単体を実装する際の開発効率については，あまり考慮していないように見受けられる．Lambda では，コンポーネントベース OS のように柔軟に構造を変更できることに加え，コンポーネント単体をプロセスと

して実行し，保護を活用して開発，デバッグを行うことが可能である．さらに，開発が終了した後には，保護をとりはらい高速化することが可能である．

8. まとめ

本研究では，OS の保守性や開発効率を向上させるため，保護機構を重視した，組込み向けマイクロカーネル OS Lambda の実装を行っている．本稿では組込みシステムにおける保護の重要性について説明を行った．また，組込みシステムにおいて，開発支援のための保護の考え方について述べ，開発支援のための保護の場合には，保護を無効にし，性能の向上や省資源を目指すべきであることを述べた．本稿ではそのような場合において，プロセス間通信を関数呼び出しに置換するモノリシック化の手法を提案，実装し，性能の評価を行った．モノリシック化手法の性能評価のために，プロセスにまたがる RPC，同プロセス内での RPC，およびモノリシック化 RPC の機能呼び出し時間を測定し，同プロセス内での RPC で 20%，モノリシック化 RPC においては 10 倍以上もの性能の向上が確認できた．

謝辞 本研究の一部は，科学技術振興事業団（JST）の戦略的基礎研究推進事業（CREST）「高度メディア社会の生活情報技術」プログラムの支援によるものである．

参考文献

1) Liedtke, J.: Improving IPC by Kernel Design, *Proc. 14th SOSP*, pp.203–205 (1993).
2) 高田広章：μ ITRON 仕様へのメモリ保護導入，第 3 回組込みシステム技術に関するサマーワークショップ予稿集 (2001).
3) Lepreau, J.: In-Kernel Servers on Mach 3.0: Implementation and Performance, *Proc. USENIX Mach III Symposium*, pp.39–56 (1993).
4) 日立製作所：日立 SuperH RISC Engine SH7708 シリーズハードウエアマニュアル．http://www.hitachi.co.jp/Sicd/Japanese/Products/micom/shmicom.htm
5) W.B. スルヤント：80386 の使い方，オーム社 (1987).
6) Bershand, B., Anderson, T., Lazowska, E. and Levy, H.: Lightweight remote procedure call, *Proc. 12th ACM Symposium on Operating Systems Principles*, pp.102–103 (1989).
7) Stankovic, J.A.: VEST: A Toolset For Constructing and Analyzing Component Based Operating Systems For Embedded and Real-Time Systems, Technical Report No.CS-2000-19, Dept.of Computer Science, Univ.of Virginia (2000).
8) Reid, A., Flatt, M., Stoller, L., Lepreau, J. and Knit, E.E.: Component Composition for Systems Software, *Proc. 4th Symposium on Operating Systems Design and Implementation*, pp.347–360, Usenix Association (2000).
9) Frohlich, A.A.: Tailor-made Operating Systems for Embedded Parallel Applications, *Proc. 4th International Workshop on Embedded HPC Systems and Applications*, No.1586, pp.1361–1373 (1999).

(平成 13 年 12 月 20 日受付)
(平成 14 年 4 月 16 日採録)

久住 憲嗣（学生会員）
1977 年生．1999 年信州大学工学部情報工学科卒業．2001 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．同年九州大学大学院システム情報科学府博士後期課程．オペレーティングシステム，組込みシステム，計算機アーキテクチャ，ネットワークに研究的関心を持つ．

北須賀輝明（正会員）
1971 年生．1993 年京都大学工学部情報工学科卒業．1995 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．同年シャープ（株）入社．パーソナルコンピュータの開発に従事．2001 年九州大学大学院システム情報科学研究院助手．モバイルコンピューティング，組込みシステム，並列/分散処理，コンパイラ，計算機アーキテクチャに研究的関心を持つ．

中西　恒夫（正会員）

1970 年生．1993 年大阪大学工学部通信工学科卒業．1998 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．博士（工学）．1996 年日本学術振興会特別研究員．1998 年奈良先端科学技術大学院大学情報科学研究科助手．2002 年九州大学大学院システム情報科学研究院助教授，また同大学情報基盤センターならびにシステム LSI 研究センターを併任．コンパイラ，計算機アーキテクチャ，組込みシステム，並列/分散処理に研究的関心を持つ．ACM，IEEE Computer Society，日本バイオインフォマティクス学会各会員．

福田　　晃（正会員）

1954 年生．1977 年九州大学工学部情報工学科卒業．1979 年同大学院修士課程修了．同年 NTT 研究所入所．1983 年九州大学大学院総合理工学研究科助手．1989 年同大学助教授．1994 年奈良先端科学技術大学院大学情報科学研究科教授．2001 年より九州大学大学院システム情報科学研究院教授．工学博士．オペレーティング・システム，コンパイラ，組込みシステム，モーバイルコンピューティング，計算機アーキテクチャ，並列/分散処理，性能評価等の研究に従事．本学会平成 2 年度研究賞，平成 5 年度 Best Author 賞受賞．著書『並列オペレーティングシステム』（コロナ社），訳書『オペレーティングシステムの概念』（共訳，培風館）．ACM，IEEE Computer Society，電子情報通信学会，日本 OR 学会各会員．