修 士 論 文

オペレーティングシステムの最適化に関する研究

指導教官　中島達夫 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

寺田 徹

1997 年 2 月 14 日

要旨

マイクロカーネルをベースとするシステムは、カーネル/ユーザ境界を越えるデータや制御の移動を頻繁に行うために、モノリシックなオペレーティングシステムと比較するとパフォーマンスが低い問題がある。これを改善するための研究として、アプリケーションのコードの一部を動的にカーネルへダウンロードすることでカーネル/ユーザ境界の移動を削減するアプローチが有力である。カーネルへダウンロードするコードは他のアプリケーションに影響を与える危険のあるエラーを含んではならない。しかし、拡張コードのエラーを完全にチェックする機構は現在まで存在しない。したがって、拡張コードの設計には簡潔で汎用的なモジュールの組み合わせが良いと考える。しかしモジュール化設計のプログラムは、単一の目的に特化して設計したものと比較すると実行の効率が劣る。そこで本研究では、モジュール化設計の効率に関する欠点を改善する最適化を提案する。

モジュール化設計のプログラムの実行効率が悪い原因の一つは、それぞれ独立して設計した複数のモジュールが、同一のデータ領域に対しての処理を含むことである。データのレジスタへのロード、何らかの処理、メモリへのストアという通常の処理の流れの中で、同じデータを複数のモジュールでそれぞれロード/ストアすることが冗長である。そこで、複数モジュールにおける処理を単一のロード/ストア間にまとめる機構とインタフェースを提供して、実行効率を向上する。

また別の原因は、モジュール集合がその外部の要因によって実行を開始するときなどに存在する。アプリケーション全体に意味をなす処理を行う前に実行を終了する場合、その状況判定以前に不要な処理を含むことが多い。意味のある実行の場合であっても、複数のモジュールで同じ判定を含むことは冗長である。そこで、ここにパケットフィルタのようなフィルタリングの概念を導入する。各モジュールは実行順序の早いモジュールに自らの実行条件を表すフィルタをいくつか渡し、そこで条件判定を行うことによって効率を向上する。さらに条件判定を受け側のモジュールで行う必要が無い場合には、より実行順序が早いモジュールにこれを伝播することによってさらなる効率化をする。

これらの最適化を動的コード生成 (Dynamic Code Genaration) を使って実現し、最適化の有効性をアプリケーションがカーネルへ拡張コードをダウンロードすることの有効性とあわせて検証する。

目 次

1 はじめに **1**

2 マイクロカーネルオペレーティングシステムをベースとするシステムの問題点 **4**

2.1 マイクロカーネル 4

2.2 柔軟性および安全性と増大する処理とのトレードオフ 5

2.3 従来の拡張可能オペレーティングシステムの問題 6

3 関連研究 **8**

3.1 拡張可能なオペレーティングシステム 8

3.2 モジュールの動的な合成 9

3.3 本研究の位置 10

4 オペレーティングシステムの拡張のためのモジュールの動的な合成 **11**

4.1 データ処理の統合 11

4.1.1 カーネル/ユーザ間でのデータ移動の指定 11

4.1.2 メモリ/レジスタ間のデータ移動の削減 12

4.2 制御の流れの最適化 13

4.2.1 コンテキストスイッチの削減 13

4.2.2 フィルタ導入による冗長な処理の削減 14

4.2.3 フィルタの伝播 15

5 実装 **18**

5.1 VCODE システム 18

5.2 モジュールの構造 19

5.3 モジュールの動的な合成 20
5.3.1 モジュールの生成、連結 20
5.3.2 モジュールの合成 20
5.3.3 モジュールの実行 24

6 評価 **25**
6.1 削減可能な制御移動のコスト 25
6.2 データ処理の統合による効果 27
6.3 フィルタの伝播による効果 28
6.4 動的合成モジュール導入のコスト 30
6.5 議論 32

7 まとめと今後の課題 **33**
7.1 まとめ 33
7.2 今後の課題 33

A インタフェース **38**
A.1 システムコール 38
A.2 データ処理コード生成インタフェース 39

第 1 章

はじめに

オペレーティングシステムの研究あるいは実用の歴史は、常にその機能を追加し続けた。その結果、オペレーティングシステムは大きく複雑になった。しかしいかに複雑になろうと、あらゆるアプリケーションにとって十分な機能と十分なパフォーマンスを提供できるオペレーティングシステムは存在しない。それは個々のアプリケーションによって要求が大きく異なるためであり、あらゆるアプリケーションの要求を満たすオペレーティングシステムは今後も現れないであろう。さらにはアプリケーションの要求が今後さらに多様化することも予想できる。

そこで近年、オペレーティングシステムの研究は単なる機能追加とは異なる方向を目指すようになった。それは、サポートしなければならないアプリケーション集合によって、あるいはその集合の変化によって、振る舞いを変える柔軟なオペレーティングシステムである。言い換えれば、アプリケーションによって拡張できるオペレーティングシステムである。その中でマイクロカーネルが登場し、従来オペレーティングシステムが提供していた機能の一部をユーザレベルのサーバによって実現させた。このことによって、システムのリブートを伴わない動的な拡張、異なるサーバを使うことで異なるサービスが受けられる柔軟性、サーバの障害がそのサーバを利用しないアプリケーションには影響を与えないという安全性が実現した。

しかし一方でマイクロカーネルはアプリケーション、カーネル、サーバの間における通信のオーバヘッドが大きいという欠点を持つ。これを改善する研究が行われる中で、アプリケーションのコードをオペレーティングシステムにダウンロードするというアプローチが有力なアプローチの一つである。しかしながら、現在このアプローチで開発されている

システムには以下の問題がある。

- 不完全な安全性の保証
  カーネルへダウンロードするコードはエラーを含んではならない。それは、カーネル内部での障害は他のアプリケーションにも影響し、マイクロカーネルがもたらした安全性を脅かすからである。カーネルへダウンロードするコードについて、現在までに文法的なチェックは可能であるが、意味的なチェックを行うことができる機構は存在しない。

- 拡張と外部との最適化の欠如
  現在開発されているシステムでは、カーネルへダウンロードするコードはそれぞれ独立にコンパイルされており、カーネルや他のダウンロードコードとの間で最適化されることは無い。

カーネルへダウンロードするコードにエラーを含まないようにするために、本研究では、カーネルへダウンロードするコードの記述にモジュール化設計を使うことによって設計を容易にすることを勧める。モジュール化設計は汎用化を促すため、再利用性も増大する。しかしながら汎用モジュールの組み合わせによるプログラムは、単一の目的に特化して設計されたプログラムよりも効率の面で劣る。そこで本研究は、モジュールの組み合わせにおける最適化を提案し、モジュール化設計の欠点である低い実行効率を向上する。

モジュール化設計のプログラムの実行効率が悪い原因の一つは、それぞれ独立して設計した複数のモジュールが、同一のデータ領域に対しての処理を含むことである。データのレジスタへのロード、何らかのデータ処理、メモリへのストアという通常の処理の流れの中で、同じデータを複数のモジュールでそれぞれロード/ストアすることが冗長である。複数モジュールにおける処理を単一のロード/ストア間にまとめる機構とインタフェースを提供する。

また別の原因は、モジュール集合がカーネル内イベントなどのモジュール外部の要因によって実行を行うときなどに存在する。モジュール集合がその実行中に、アプリケーション全体として意味のある実行状況ではないと判定して実行を終了する場合があるが、このときいくつかのモジュールの実行が終了するまでこの判定できないことがあり、その場合その実行は不要な処理を多く含む。またはその実行がアプリケーション全体として意味のある場合であっても、複数のモジュールで同じ判定を含むことは冗長である。そこで、こ

こにパケットフィルタのようなフィルタリングの機構を導入して、実行状況の判定を集約する。

これらの最適化を動的コード生成 (Dynamic Code Genaration) を利用して実現する。動的コード生成は、まずモジュール集合をひとつの関数にまとめることによってプロシージャコールのコストを削減する。また処理の順序を動的に決定するので、データ処理を統合してメモリ/レジスタ間のロード/ストアの削減を可能にする。そしてフィルタ構造をマシンコードへ変換することによって、実行時の処理を削減する。

以降の各章の概要を示す。2 章ではマイクロカーネルの問題点について述べる。3 章では拡張可能なオペレーティングシステムと、モジュールの合成についての関連研究を紹介する。4 章ではデータ統合と、フィルタリングを中心とした最適化について述べる。また、カーネルへコードをダウンロードする利点についても述べる。5 章では実装の概要を実行の流れとともに示す。6 章では基本的な評価を行う。7 章では今後の課題を示す。

第 2 章

マイクロカーネルオペレーティングシステムをベースとするシステムの問題点

2.1 マイクロカーネル

アプリケーションが多様化するにつれて、そのアプリケーションが十分な性能を発揮するためのオペレーティングシステムに対する要求も多様化する。一つのオペレーティングシステムに多様化するすべてのアプリケーションの要求を十分に満たすことを期待するのは不合理であるため、研究者はアプリケーションによって拡張することが可能なオペレーティングシステムを目指すようになった。その流れの中で Mach、Chorus、Amoeba などのマイクロカーネルに基づくオペレーティングシステムが登場した。これらのシステムでは、カーネルは、アドレススペース管理、スレッド管理、プロセス間通信、デバイス管理などの基本的な機能のみを提供し、従来はカーネルが提供していたファイルシステムやネットワークシステムなどの機能は、ユーザレベルのサーバで実行することとした。マイクロカーネルは以下のような利点をもたらした。

柔軟性　サーバとして実現する機能は同時に複数存在することができ、アプリケーションはその要求に応じてサーバを使い分けることができる。

動的な拡張　新しい機能の追加はサーバの拡張あるいはサーバの追加によって行うことができるため、システムのリブートを伴わない。またリブートを伴わないため、新しいサー

ビスの実現、インストール、デバッグが容易である。

安全性　サーバ内部で発生した障害が、そのサーバを利用しないアプリケーションには全く影響しない。

2.2　柔軟性および安全性と増大する処理とのトレードオフ

しかし、マイクロカーネルに基づくシステムには、従来のモノリシックカーネルのオペレーティングシステムと比較するとパフォーマンス上の問題がある。マイクロカーネルに基づくシステムを含めた従来のオペレーティングシステムの多くは、カーネルレベルとユーザレベルの境界にハードウエアのプロテクションを利用している。このため、この境界を越える制御やデータの移動にはコンテキストスイッチとデータのコピーが必要である。図2.1に示すように、一部の機能をユーザレベルのサーバで実現するマイクロカーネルに基づくシステムでは、その機能をカーネルに含んでいたモノリシックカーネルのオペレーティングシステムと比較すると、カーネル/ユーザ間の移動が頻繁である。このこと

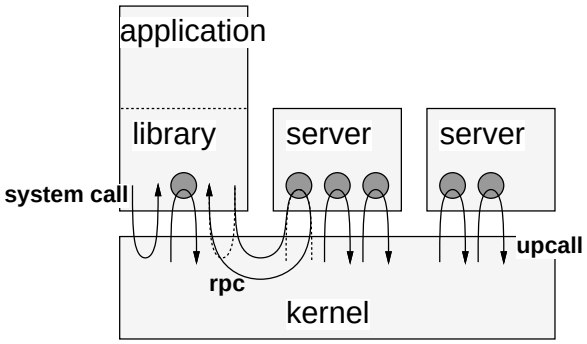


図 2.1: マイクロカーネルに基づくオペレーティングシステム

がアプリケーション全体としてのパフォーマンスを低下させている。さらに、システムの拡張はサーバの拡張や追加をすることによって実現されるので、システムを拡張するごとにパフォーマンスは低下することになる。

2.3 従来の拡張可能オペレーティングシステムの問題

このマイクロカーネルに基づくシステムのパフォーマンスに関する問題を解消する研究がいくつかなされている中で、アプリケーションが一部のコードを動的にカーネルへダウンロードして、カーネル/ユーザ境界の移動を削減するアプローチが有力である (図 2.2)。SPIN[BSP+95]、VINO[SESS94]、DKM[追川 96] などがこのアプローチを採用している。

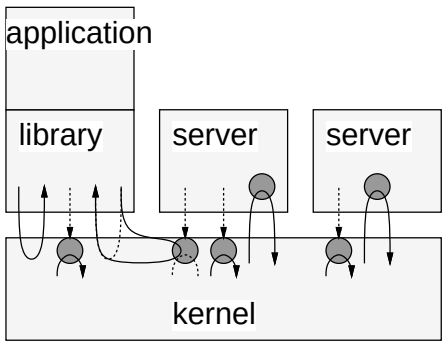


図 2.2: カーネルへのコードのダウンロードによる効率化

しかしながら、これらのシステムには以下の問題がある。

不完全な安全性の保証　カーネルレベルでアプリケーションのコードを実行することはマイクロカーネルがもたらした、拡張がその拡張を利用しないアプリケーションに影響を与えないという安全性を脅かす。これらのシステムでは、Modula3 のような型安全な言語によってカーネル、拡張部ともに記述すること、あるいは sandboxing などのソフトウェアフォールトアイソレーション [WLAG93] を使うことなどによって、カーネル内での安全性を保証する。しかしながら、ロックを長時間開放しない、などという意味的なエラーは型安全な言語の使用やソフトウェアフォールトアイソレーションだけでは防ぐことはできない。カーネルへダウンロードするコードの設計において、機能の充実や効率の向上を望むほどコードが複雑になり、エラーが発生しやすくなるという問題がつきまとう。

拡張と外部との最適化の欠如　これらのシステムではカーネルへダウンロードするコードはそれぞれ独立にコンパイルされており、カーネルや他のダウンロードコードとの間で、最適化されることは無い。したがって、コンパイル技術が発達してより最適化されたカー

ネルの実現が可能になると、機能の複数ダウンローディングよりも、機能豊富な巨大モノリシックカーネルが実行効率の点から再び重要視されることになるかも知れない。

第 3 章

関連研究

3.1 拡張可能なオペレーティングシステム

現在、オペレーティングシステム研究では、アプリケーションによって動的に拡張できるシステムが多い。その目的はパフォーマンスの向上と機能の追加である。マイクロカーネルアーキテクチャは、従来カーネルが提供していたサービスの一部をユーザレベルで実現することによって動的な拡張を可能にした。現在はその次の段階として、マイクロカーネルアーキテクチャの弊害であるプロテクションドメイン間の移動のコストを削減する研究が中心である。

そのためのアプローチは大きく分けて二種類に分類できる。一つはアプリケーションの機能をカーネルへ移すもの、もう一つはマイクロカーネルのアプローチをさらに押し進めてオペレーティングシステムはハードウェアのアブストラクトのみを提供するというものである。

SPIN　SPIN[BSP+95] はオペレーティングシステムのインタフェースと実装をアプリケーションが拡張するための基盤と基本的なサービスの集合を提供する。アプリケーションの設計者は、基本的なコアサービスのサブクラスとして拡張サービスを構築することによってアプリケーション固有のインタフェースを実現する。拡張は動的にカーネルとリンクする。カーネルと拡張は Modula-3 を使って記述し、カーネル内での安全性を保証する。

VINO　VINO[SESS94] はアプリケーションのコードをオペレーティングシステムにダウンロードしてその振る舞いを変えるシステムである。安全性の保証にはsandboxing[WLAG93] を使用している。拡張の障害からの回復を容易にするため、VINO は拡張をトランザクションで実行する。ファイルを使うことによって拡張はシステムのリブート後にも存在する。

DKM　DKM(Dynamic Kernel Module)[追川 96] はマイクロカーネルの拡張の単位である DKM をカーネルへロードするソフトウェアアーキテクチャを提案する。DKM はメソッドと、メソッド以外による操作を禁じるメンバからなるオブジェクトである。DKM のロード、デタッチは DKM サーバが責任を持つ。DKM サーバは DKM 間の関係を把握し、カーネル内の DKM の状態を絶えず注意する。また DKM サーバは動的適応のためのイベント処理を集中管理し、DKM からイベント処理を分離する。

インタプリタによる拡張　カーネルにコードを挿入する実装の一つに、インタプリタ言語を使って拡張を記述し、そのインタプリタをオペレーティングシステムに組み込む方法がある。Pathfinder[BGP+94] などのネットワークのパケットフィルタの多くはこの実装を使っている。HiPEC[LCC94] はインタプリタを使って仮想メモリのキャッシングポリシを制御する。しかしこれらは言語の制約上、拡張できるのは一部の機能だけである。

Exokernel　Exokernel[EKJ95] はマイクロカーネルのアプローチをさらに押し進めたものと考えることができる。カーネルからアブストラクションを取り除き、低レベルなインタフェースをアプリケーションに提供する。カーネルは物理デバイスのアクセス制御のみを行う。アプリケーションが要求するアブストラクションはユーザレベルのライブラリによって実現する。

3.2　モジュールの動的な合成

モジュール化設計における複数モジュールのデータ処理の統合は、ネットワークの分野でいくつか研究がなされている。

プロトコルレイヤの統合　Abbott と Peterson[AP93] は複数プロトコルでのデータ処理を統合するシステムを提案した。このシステムはコンパイル時にパイプと呼ぶマクロを、統

合したループに合成する。パイプとは数バイトの入力を受け数バイトの出力を返す計算である。

ASHs　ASHs[WEK96][EWK96](Application-specific safe message handlers) はネットワークメッセージの到着時の実行のためのコードをカーネル内に挿入するためのシステムである。ASHs はプロトコルの動的な合成のためのサポートを提供する。レイヤ間での処理の統合は Abbott と Peterson のシステムと同様にパイプを使う。メッセージのユーザ空間へのコピーも統合するので効率が良い。動的な合成を実現するために、動的コード生成として VCODE システムを使う。

3.3　本研究の位置

拡張可能なオペレーティングシステムはいくつかの方式で行われているが、その中でカーネルへアプリケーションがコードをダウンロードして拡張を行うシステムがやや有力であると考えられる。しかし、ダウンロードするコードの組み合わせについて述べている研究はあまり無い。本研究は設計の容易さと実行効率の良さの両立を目指し、ダウンロードコードの組み合わせにおいての最適化を提案する。本論文ではカーネルへダウンロードする方式を採用する場合について述べているが、提案する最適化の手法はカーネルの機能をさらにユーザレベルに移す方式にも利用することができる。

モジュール化プログラムの合成はネットワークの分野でいくつかの研究があるのみであるが、本研究はより広い応用を視野に入れる。

第 4 章

オペレーティングシステムの拡張のためのモジュールの動的な合成

マイクロカーネルをベースとするシステムのパフォーマンスを改善するアプローチとして、本研究においてもアプリケーションの一部のコードをカーネルに動的にダウンロードすることを支持し、本章でその利点を述べる。カーネルへダウンロードするコードは他のアプリケーションに影響を与えるようなエラーを含んではならない。このことより、カーネルにダウンロードするコードの作成にはモジュール化設計が望ましい。それは、少機能のモジュールの組み合わせによって求める機能を実現することによって、個々のモジュールの設計は簡略になりエラーが混入する可能性が低くなるからであり、また、モジュール化することで個々のモジュールは汎用性が増し、プログラムの再利用性を良くするからである。しかし、汎用モジュールの組み合わせは、単一目的に最適化したモジュールと比較すると効率の面で劣る。そこで、このモジュール化設計の欠点を補う最適化について本章では述べる。

4.1 データ処理の統合

4.1.1 カーネル/ユーザ間でのデータ移動の指定

Mach のようなマイクロカーネルアーキテクチャではタスクとカーネルとの通信をメッセージとポートによって実現している。受信タスクは受信ポートにキューイングされた

メッセージを、自身のアドレス空間のバッファにコピーしてこれを使用する。ここで、受信タスクが必要とするデータがメッセージのうちのわずかな部分である場合、不要なデータコピーが行われている。また、ネットワークサブシステムやファイルシステムなどのサーバを複数利用するアプリケーションでは、サーバが増えるごとにその間でのデータの移動が発生するが、それはすべてカーネル内のポートを介すため、コピーのコストが大きい。これを解消するために、アプリケーションがカーネル内でデータパスを作成し、カーネル内データへ一般的な処理を行うことを可能にする。また、最終的にカーネル内データのどの部分をユーザ空間のどこにコピーするかもアプリケーションが制御する。作成したデータパスは効率の良いシステムコールとなる。

4.1.2 メモリ/レジスタ間のデータ移動の削減

ソフトウェアのモジュール化は設計の効率が良い反面、実行の効率が悪い。その大きな要因は、それぞれ独立して設計した複数のモジュールが同一のデータ領域に対して処理を行うためである。あるモジュールでレジスタにロードし、何らかの処理を行い、メモリにストアしたデータ領域を、再び別のモジュールでロード、処理、ストアする。このロード/ストアのコストがモジュール化設計ソフトウェアの実行効率を落している。そこでこれを改善するために、複数モジュールでの処理を一組のロード/ストア間にまとめる機構とインタフェースを提供する。これを実現するためには動的コード生成を利用する。

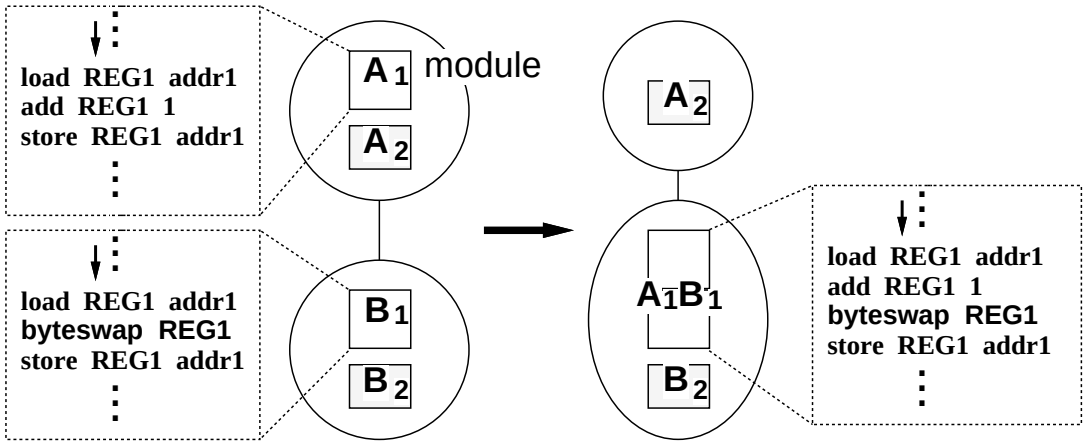


図 4.1: load/store の削減

現在の一般的な計算機のアーキテクチャでは、処理のオペランドのサイズが単一ではなく、また、そのため複数のモジュールが処理を行う同じデータ領域を異なる区間としてロード/ストアを行う場合があり、処理の統合を困難にする。例えば図 4.2で、データ領域

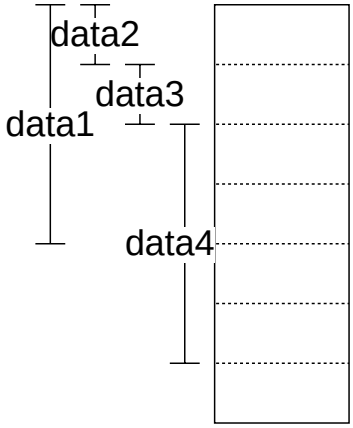


図 4.2: 処理区間の重複

data1、data2、data3、data4 の順に処理を行う場合を考える。data1 と data2 のアドレスは等しいためその統合は容易であるが、data3 や data4 のアドレスは異なるために統合が困難である。すべての処理を data2 のサイズで行うことにすれば data1 の一部と data2、data1 の一部と data3、data1 の一部と data4 の一部の統合が可能になる。しかしながら、これは通常一インストラクションで行う data1 の処理を四回に分けてで行うことになる。そこで、現在の実装ではこのような場合の処理の統合を行わない。図のような場合には、先頭から二つの処理のみを統合する。data3 はオペランドサイズを変更し、シフト命令を使うことによって先頭から二つの処理と統合することができるが、現在の実装はこれも省略している。

4.2 制御の流れの最適化

4.2.1 コンテキストスイッチの削減

アプリケーションがモジュールをカーネル内にダウンロードすることは、4.1 で述べたようにデータパスを作ることを可能にすると同時に、コンテキストスイッチの数を削減し

てパフォーマンスを向上する。

さらに、各モジュールにおけるデータ処理を動的コード生成によって一つの関数にまとめることによって、関数呼び出しのコストを削減する。

4.2.2 フィルタ導入による冗長な処理の削減

モジュールの組み合わせによるソフトウェアは、特化したモノリシック構造のソフトウェアと比較すると冗長な処理を多く含むことになる。例えば、図 4.3で、制御の流れが図の上のモジュールから下のモジュールへ順に移るような処理をカーネル内に作成するとする。通常、各モジュールは受け取ったデータに関して自らの処理を実行すべきである

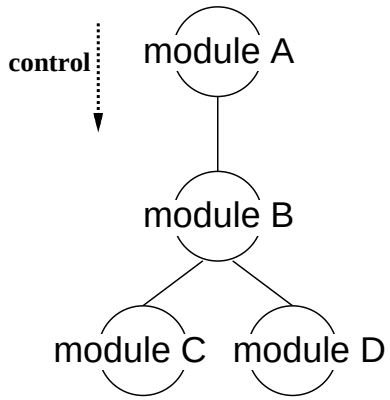


図 4.3: モジュール化設計における制御の流れ

かどうかの条件を持っている。ここでモジュールをそれぞれ独立して設計することを可能にするため、どのような連結をするかの知識を各モジュールは持たない。したがってモジュール B はモジュール C とモジュール D の両方に制御を移さなければならない。モジュール C とモジュール D で行う条件判定のうち、同じものが存在すると同じ処理を二度行うことになる。また、モジュール C とモジュール D の実行条件が全く異なる場合であっても、モジュール C やモジュール D の条件判定においてモジュール集合としての処理を終了する場合、そのときまでアプリケーションにとっては意味のある処理を行っていないことがある。このときモジュール A やモジュール B で行った処理は無駄と言うことになる。モジュール C とモジュール D の条件判定がそれぞれが異なるブロックに存在する場合には、キャッシュミスによるパフォーマンス低下の可能性もある。

そこで、ここにネットーワークのパケットフィルタのようなフィルタの機構を導入する。フィルタは制御を渡してもらう前置連結モジュールに各モジュールが動的に渡す実行

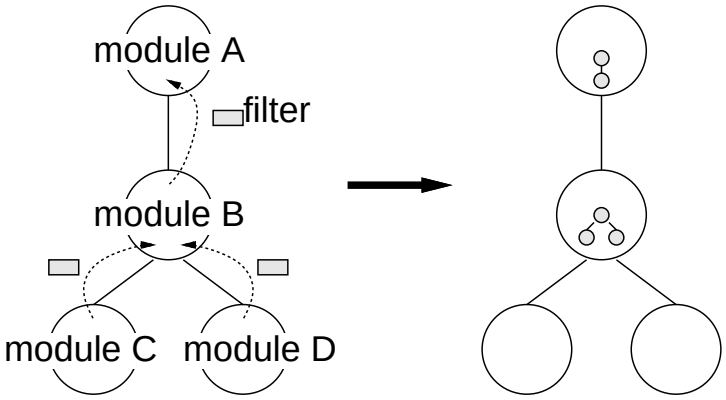


図 4.4: フィルタ

条件とする。現在の実装での実行条件とは、前置連結モジュールでデータ処理ルーチンが終了したときに、前置連結モジュールが参照しているデータとのパターンマッチとする。フィルタを受け取る側のモジュールは、パケットフィルタの実装で行われているように、受け取ったフィルタからカーネル内データを参照して分岐を行う木を作ることによって重複する条件判定処理を削減する。実行の時間効率を向上するために、この分岐のための木から動的コード生成によって、モジュール A やモジュール B でのデータ処理後に制御を移すモジュールを決定するマシンコードを生成する。生成するマシンインストラクションコードはフィルタ条件一つ一つのに対応する条件比較の対象を即値オペランドとして含むことができる。このことは、分岐木そのものをたどることと比較して、木をたどるポインタ参照や比較対象となる値の読み込みが無い分、パフォーマンスを向上する。また、この分岐によってはモジュール C やモジュール D、あるいはモジュール B へ制御を移す必要が無くなり、キャッシュミスの可能性も低くなる。

4.2.3 フィルタの伝播

冗長な処理の削減とメモリの局所化をさらに進めるためにフィルタの伝播を可能にする。図 4.5のようなモジュール連結において、モジュール C またはモジュール D に制御の流れが到達してのみ、ユーザ空間にデータの一部をコピーするなどの、アプリケーショ

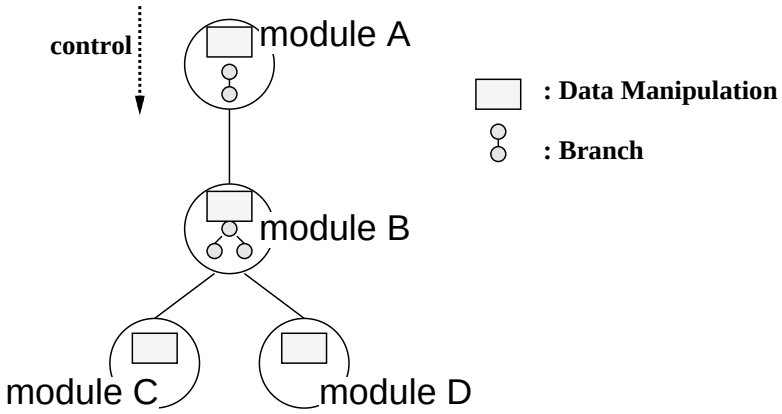


図 4.5: フィルタ伝播前

ンにとって意味がある振る舞いをするとする。この場合、各モジュールでそれぞれデータ処理と分岐の処理を行っている。モジュール C とモジュール D が渡したフィルタからモジュール B に生成した分岐コードによって処理が終了する場合にも、そこまでの処理すなわちモジュール A のデータ処理、モジュール A の分岐、モジュール B のデータ処理はすべて行われる。

フィルタの伝播を可能にすることで冗長な処理を削減する。先に述べたように、フィルタリングは参照データとのパターンマッチによって行う。ここで、自らのデータ処理ルーチンで、受け取るフィルタに関連するデータフィールドに全く関与しない場合がある。この場合にモジュールは受け取ったフィルタをさらに制御の流れの順で一つ手前のモジュールに渡すことができることとする (図 4.6)。例えば、モジュール B が参照データのうち、モジュール D から受け取るフィルタと比較するフィールドへの書き込みを行っていないとすると、分岐処理はモジュール B のデータ処理の前に行うことができる。すなわち行ってもアプリケーションにとって意味をなさないような、各モジュールでのデータ処理を削減することができる。図 4.6はモジュール D が渡すすべてのフィルタをモジュール B がモジュール A に伝播できる様子を示している。すべてのフィルタを伝播できる場合にはモジュール B とモジュール D を一つのモジュールと考えることができる。

また、図 4.6では、フィルタを伝播したときのメモリの局所性の向上も示している。モジュール B とモジュール D が生成するデータ処理 b と d のインストラクションについての局所性が増しており、キャッシュミスが起こる可能性を低くすると期待できる。連結す

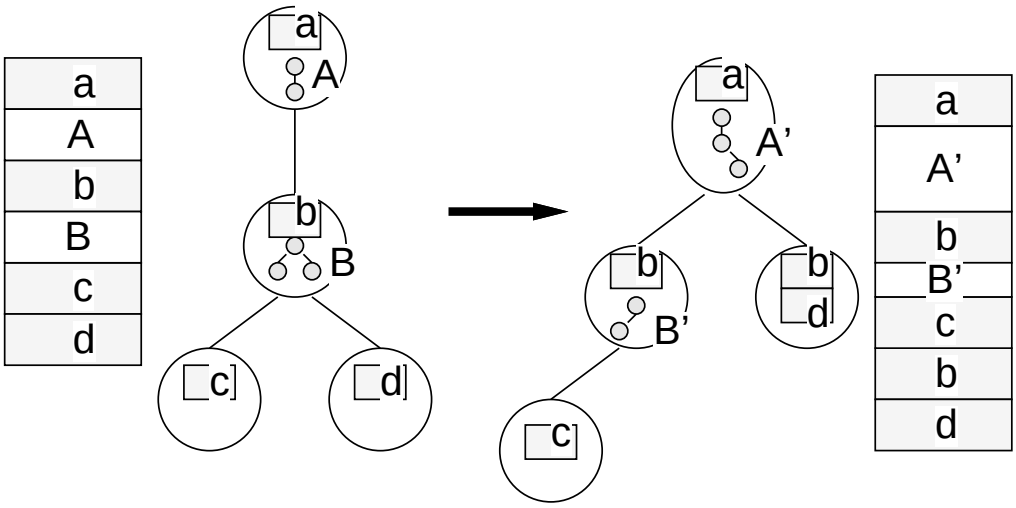


図 4.6: フィルタ伝播

るモジュールの数が増えるほど、フィルタ伝播をするときとしないときのメモリの局所性の差は大きくなる。

第 5 章

実装

4 章で述べた、モジュールの動的な合成の有効性を示すための実装を行った。Real-Time Mach のカーネルの拡張を行って、システムコールを追加した。

以降の各章ではアプリケーションの設計者が指定する単位としてのモジュールと、合成を行って最終的に生成する関数とを特に明確に区別したい場合に、前者をスペシフィケーションモジュール、後者をインプリメンテーションモジュールと呼ぶ。

5.1 VCODE システム

連結するモジュール間での関数呼び出しの削減、レジスタへのロード/ストアの削減のための処理の統合、フィルタによる動的な分岐生成を実現するために、動的コード生成 (dynamic code generation) を用いる。動的コード生成には Massachusetts Institute of Technology の Parallel and Distributed Operating Systems Group で開発された VCODE システム [Eng96] を利用する。

VCODE はコード生成の高速化と、生成したコードの効率の良さを両立することを目的としている。VCODE はこの目的を達成するために、マシンコード生成のための中間データ構造を使わずに、適所に直接コードを生成するという特徴を持つ。実行時においての中間表現の生成や使用の必要が無いことは、VCODE を効率的にするだけではなく、拡張性も良くしている。

VCODE は RISC のインターフェースに似た、低レベルでマシン独立なインタフェースを持つ。このインタフェースはマクロとして実現されており、実行時には直接マシンコー

ドに変換されてインストラクションバッファとして用意したメモリ位置に書き出される。低レベルなインタフェースは、レジスタ割り当てなどを静的なコンパイル時にさせる、コンパイラのサポート無しに速いコード生成を可能にする、プリフェッチや分岐予測など高レベル言語では記述できないことが記述できる、という利点を持つ。

本研究で動的コード生成システムとして VCODE システムを採用した理由は、インタフェースを記述したそのままの順に直接マシンコードが生成される点である。このことによって、処理の統合や分岐コード生成を容易に行うことができた。VCODE は現在、MIPS、SPARC、Alpha の各プロセッサで動作するものが配布されている。これを x86 アーキテクチャで利用するため移植を行った。

5.2 モジュールの構造

以降の節での説明のため、図 5.1にモジュールの構造体を、表 5.1にシステムコールを示す。

```
typedef struct _dmc_module {
        module_link *prior;             /* 前置連結モジュール */
        module_link *posterior;         /* 後置連結モジュール */
        XPFPattern *requirement;        /* フィルタ */
        void *reference;                /* 参照データ */
        dmc_manipulate *manip;          /* データ処理 */
        dmc_integrate *integ_list;      /* 統合処理エントリ */
        v_label_type label;
        int compile_state;
        int type;
        int propagate_type;             /* フィルタ伝播条件 */
} dmc_module;
```

図 5.1: スペシフィケーションモジュール

kern_return_t dmc_module_create(specid, req, type, propagatetype)	スペシフィケーションモジュールの生成
kern_return_t dmc_manipulate_set(specid, manipid, arg, argsize)	データ処理コード生成関数のスペシフィケーションモジュールへの関連づけ
kern_return_t dmc_module_connect(prior, posterior)	スペシフィケーションモジュールの連結
kern_return_t dmc_module_compose(implid, specid, name)	インプリメンテーションモジュールの生成
kern_return_t dmc_module_exec(implid)	インプリメンテーションモジュールの実行
kern_return_t dmc_module_destroy(implid)	インプリメンテーションモジュールの削除

表 5.1: システムコール

5.3 モジュールの動的な合成

モジュールの動的な合成の流れにそって実装の概要を述べる。

5.3.1 モジュールの生成、連結

モジュールの設計は大きく分けて、二つに分けられる。一つはフィルタの記述、もう一つはカーネル内データ処理の記述である。フィルタの記述はパケットフィルタと同様に宣言的に行い、フィルタ構造体のリストとしてモジュールが持つ。カーネル内データ処理は vcode で記述する。システムコール dmc_module_create は引数にフィルタ構造体を取り、カーネル内にスペシフィケーションモジュールを生成する (図 5.2)。dmc_manipulate_set はデータ処理生成コードをスペシフィケーションモジュールのメソッド manip と関連づける。dmc_module_connect は、カーネル内モジュールを連結する (図 5.3)。

5.3.2 モジュールの合成

フィルタの伝播について、そのモジュールに関与しないフィルタは、それがある一つの後置連結モジュールから受け取ったフィルタ集合の一部であっても、その一部のみすべて伝播するのが理想的である。しかし現在の実装では、一つの後置連結モジュールから受け

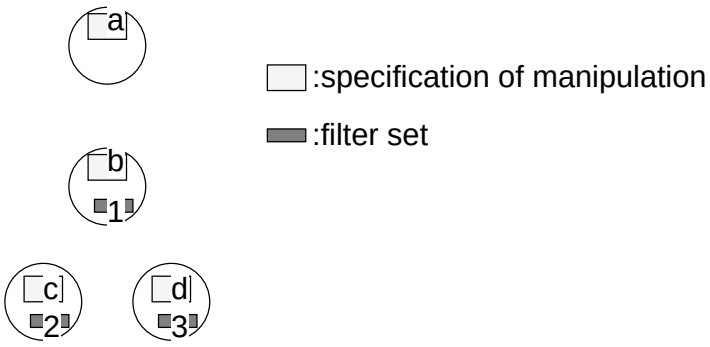


図 5.2: スペシフィケーションモジュールの生成

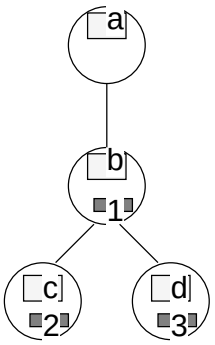


図 5.3: スペシフィケーションモジュールの連結

取ったフィルタ (通常は複数) をすべて伝播できるときのみに行うこととしている。その理由は、関与の有無を調べるためのコストが膨大であると予想できることと、すべて伝播したときの効果が明確ではない段階で、一部のみの伝播による効果がそれほど期待できないことである。また、現在の実装では伝播ができるかどうかをモジュールのタイプによって判断する。一つのモジュールから受けたすべてのフィルタを伝播できるときには、フィルタを渡すモジュールと受けるモジュールを一つのスペシフィケーションモジュールと考えることができる。

モジュールの合成はシステムコール dmc_module_compose によって行う。まずすべてのフィルタを伝播できるときには一つのモジュールと考えられることから、モジュールの連結構造を変更する (図 5.4)。

その後データ処理のコード生成と分岐のコード生成を開始モジュールから順に連結した

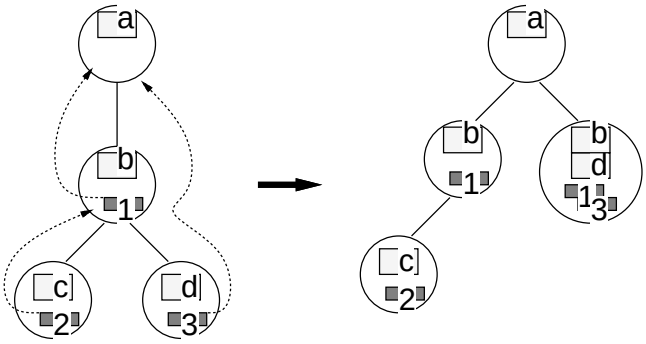


図 5.4: フィルタ伝播によるモジュールの連結構造の変更

すべてのモジュールを探索して行う。以下にそれぞれの実装について説明する。

データ処理コードの生成　データ処理コード生成はモジュールのメソッド manip によって行う。メソッド manip は vcode で記述し、実行時にインストラクションバッファへマシンコードを生成する (図 5.5)。メソッド manip 中で、データ処理の統合のためにイン

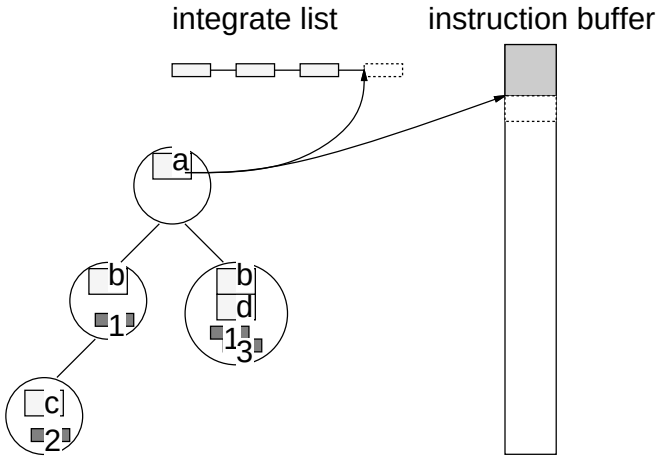


図 5.5: データ処理コードの生成

タフェース dmc_manip_integrate を使って記述された部分は、即時にインストラクションバッファへのコード生成を行わず、モジュールのメンバであるリスト integ_list に追加して、コード生成を遅延する。リストに連結した処理は、その遅延のポリシや処理を行う

データ領域、処理のサイズなどに応じて最も適当なタイミングにインストラクションを生成する。遅延のポリシは現在のところ、できる限り遅延、遅延しないの二種類である。

現在の実装ではメソッド manip のカーネルへの動的なリンクは行っていないが、今後 DKM[追川 96] などを使うことによって、任意のデータ処理ルーチン生成関数をアプリケーションが指定できるようにする予定である。

分岐コードの生成　分岐のコード生成はまず後置連結モジュールからフィルタ集合を受け取ることから始める。受け取ったフィルタから分岐のための木を生成する。図 5.6にその簡略図を示す。この木の生成には Pathfinder[BGP+94] を利用しており、同じ比較を何度も行わない最適化をする。Pathfinder ではパケットの到着時にこの木をたどってパケット

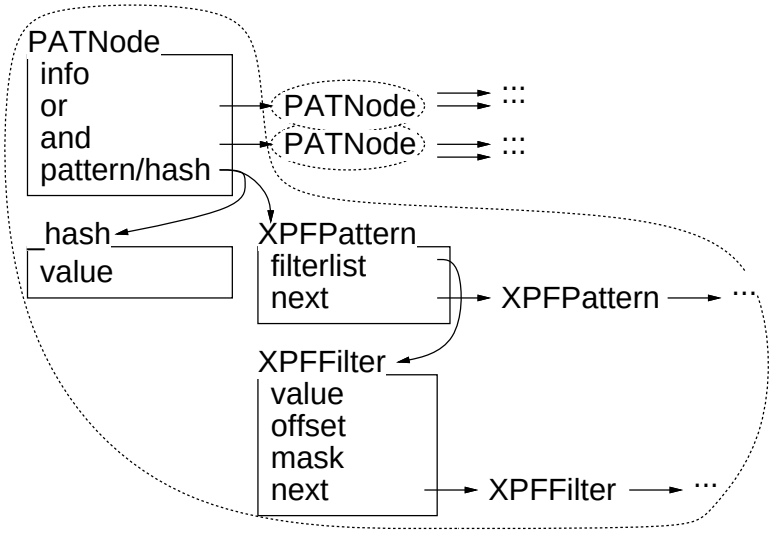


図 5.6: データ解釈のための木

の解釈を行うが、パケットのデータと比較を行うデータを木から取り出すオーバヘッドが大きい。そこで、生成コード実行時に Pathfinder のデータ解釈の関数を用いることはせず、コード生成時に木をたどって分岐のためのマシンコードを生成する (図 5.7)。このことによって、実行時に木をそのまま利用する場合には、

```
filter1 = patnode1->common.pattern->filterlist;
if(filter1->value == data[filter1->offset] & filter1->mask) ···
```

というような処理が必要なところを、

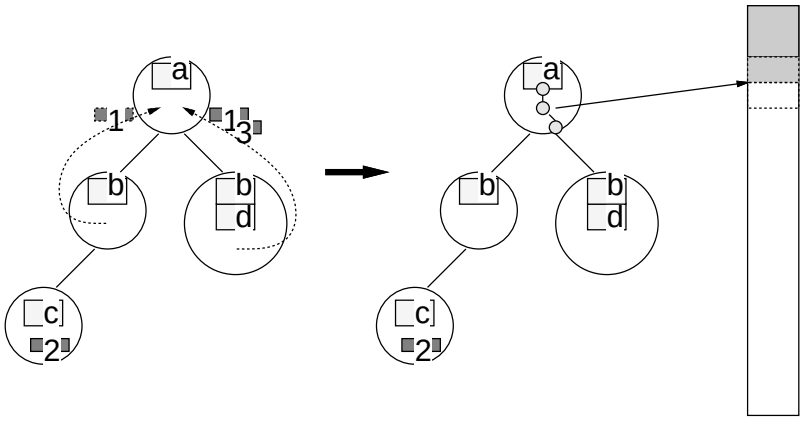


図 5.7: 分岐コードの生成

```
v_ldii(register, &data[patnode1->common.pattern->filterlist->offset]);
v_cmpii(register, register, patnode1->common.pattern->filterlist->value);
```

のようにして、比較の対象となるデータを即値オペランドとして生成するインストラクションに組み込むことによって、実行効率を改善する。一つの vcode インストラクションはほとんどマシンインストラクション一つに対応しているので、大きく効率を改善することができる。

すべてのモジュールのデータ処理コード、分岐コードの生成後、モジュールのラベルによってラベリングをしたインストラクションバッファ内のラベル間でジャンプのアドレス解決を行い、インプリメンテーションモジュールの生成を終了する。生成するコードの局所性は連結したスペシフィケーションモジュールを探索する順序に依存するが、現在までに深さ優先探索と幅優先探索を用意している。

5.3.3 モジュールの実行

生成したスペシフィケーションモジュールは dmc_module_exec によって、システムコールとして呼び出すことができる。

第 6 章

評価

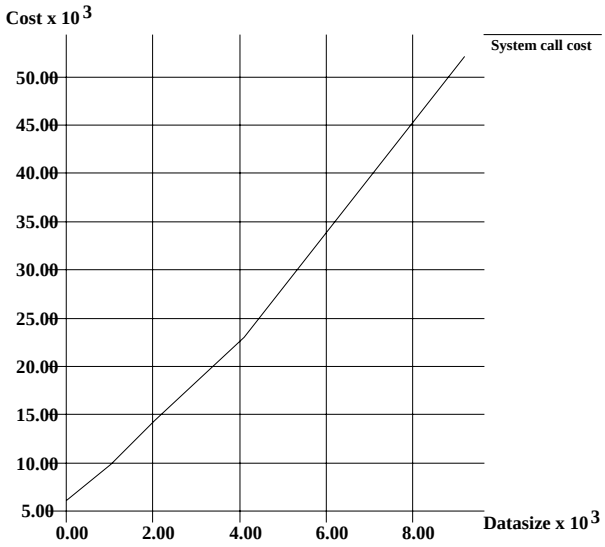
6.1 削減可能な制御移動のコスト

アプリケーションがカーネルへコードをダウンロードすること、そしてそれらを合成することによる効率の改善を示すために、まずプロシージャコール、システムコール、リモートプロシージャコールの各コストの測定を行い、アプリケーションの構成による各コストの省略によって可能となるパフォーマンスの向上を考察する。Real-Time Mach における各呼び出しのコストを表 6.1に示す。プロシージャコールとシステムコールは引数なしの NULL 関数の場合、RPC はメッセージヘッダのみの受け渡しのコストを示している。システムコールと RPC には大量のデータ移動が伴う場合があり、そのコストはデータ量によって大きく異なる。これを図 6.1に示す。横軸は呼び出し、戻りともに移動するデータの大きさを、縦軸は必要なサイクル数を示す。

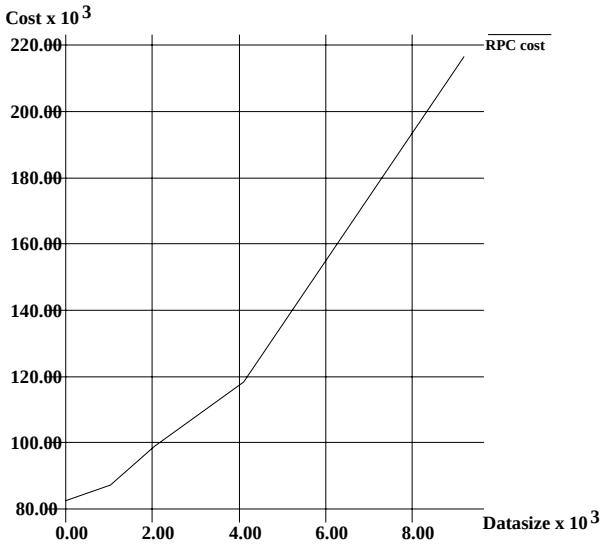
	コスト (cycle)
プロシージャコール	19
システムコール	5486
リモートプロシージャコール	82637

表 6.1: 制御に関するコスト

図 6.2(a) のような構成のアプリケーションについて考察する。図はアプリケーションが N1+N4 個、サーバが N2+N3 個の関数モジュールを含むことを示す。アプリケーショ



システムコール



リモートプロシージャコール

図 6.1: 移動するデータ量に対するコスト

ンのモジュールをサーバへ挿入することが可能ならば図 6.2(b) のような構成で、カーネルへ挿入することが可能ならば図 6.2(c) のような構成でアプリケーションを構築することができる。アプリケーション、サーバ内のモジュールを動的に合成することによってプロシージャコールの削減が可能となる。各構成においての削減可能なプロシージャコールの数は表 6.2の通りである。

図 6.2(a)	N1 - 1 + N2 - 1 + N3 - 1 + N4 - 1
図 6.2(b)	N1 + N2 - 1 + N3 + N4 - 1
図 6.2(c)	N1 + N2 + N3 + N4 - 1

表 6.2: プロシージャコールの削減

図 6.2(c) の構成はさらに RPC のコストを完全に削減することができ、大幅に効率を改善する。逆に、RPC のコストと比較するとプロシージャコールのコストは取るに足らない。図 6.2(b) の構成は図 6.2(c) と比較すると RPC のコストが大きいが、RPC のコストはメッセージの大きさによって大きく異なるので、アプリケーションとサーバ間で受け渡すメッセージを小さくするようにアプリケーションのモジュールをサーバへ挿入すると効

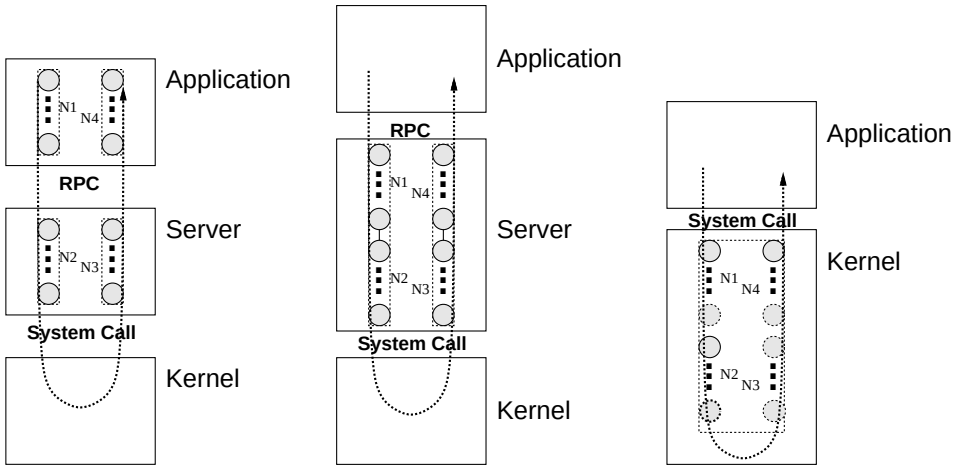


図 6.2: アプリケーション構成の例

率はある程度改善できる。

6.2 データ処理の統合による効果

モジュールの動的な合成によってデータ処理の統合を可能にした。このデータ処理の統合の効果を調べる実験として、400byte のデータ領域に 4byte ごとに書き込みを行うモジュール 1 個以上と、初期化モジュールとの合成によって生成する関数の実行サイクル数を測定する。処理の統合を行った場合と行わない場合についての結果を図 6.3に示す。処理の統合を行わない場合は 100 回ずつの書き込みを各モジュールが順に行う、統合する場合は、各 4byte ごとにモジュールが順に書き込む。グラフの横軸は書き込みを行うモジュール数を表す。実行時のインタラプトによる誤差が生じるため、測定値は 1000 試行中サイクル数が小さいもの 10 個の平均とする。グラフより一組のデータ処理を統合することによって 4〜6 サイクル短縮できることがわかる。実際削減しているのは、データ参照位置のロードとデータのロード/ストアの合計 3 サイクルであるが、それ以上の短縮が見られるのはキャッシュミスの減少と考えられる。

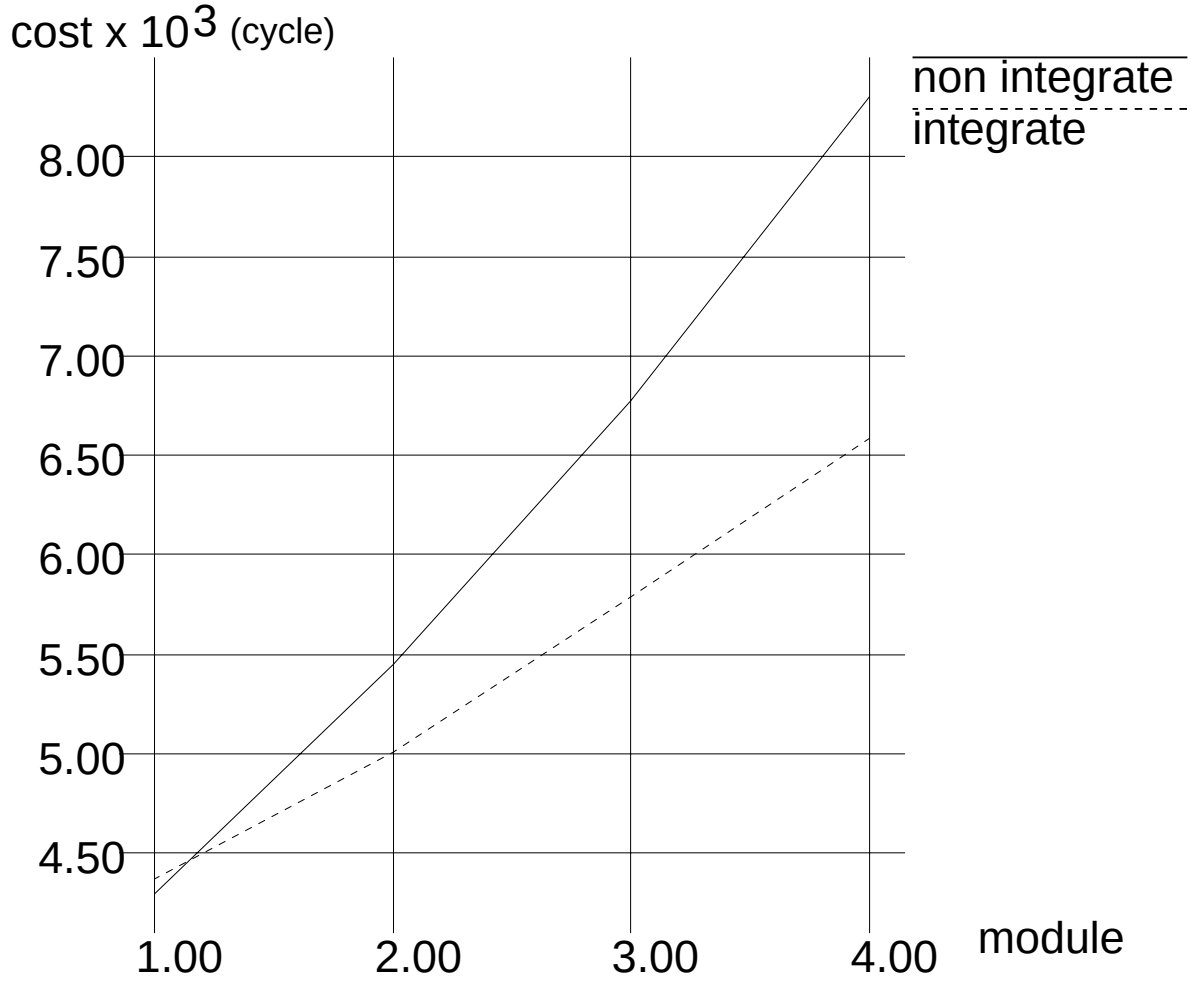


図 6.3: データ処理の統合の効果

6.3 フィルタの伝播による効果

フィルタを伝播すると、データ参照の局所化が進みキャッシュミスが減ることによって効率は向上する。この効果を調べる実験を行った。まず、データ処理を行わず、32bit のパターンマッチを一つフィルタとして持つモジュールを一列に連結し、これを合成して生成した関数の実行サイクル数を測定する。結果を図 6.4に示す。グラフの横軸は初期化モジュール以外のモジュール数、縦軸は実行サイクル数を示す。実線は伝播を全く行わなかった場合、点線は全てのモジュールにおいて伝播を行った場合を示している。両者の生成するコードの違いは、フィルタから生成する分岐コードについてのジャンプの順序、回

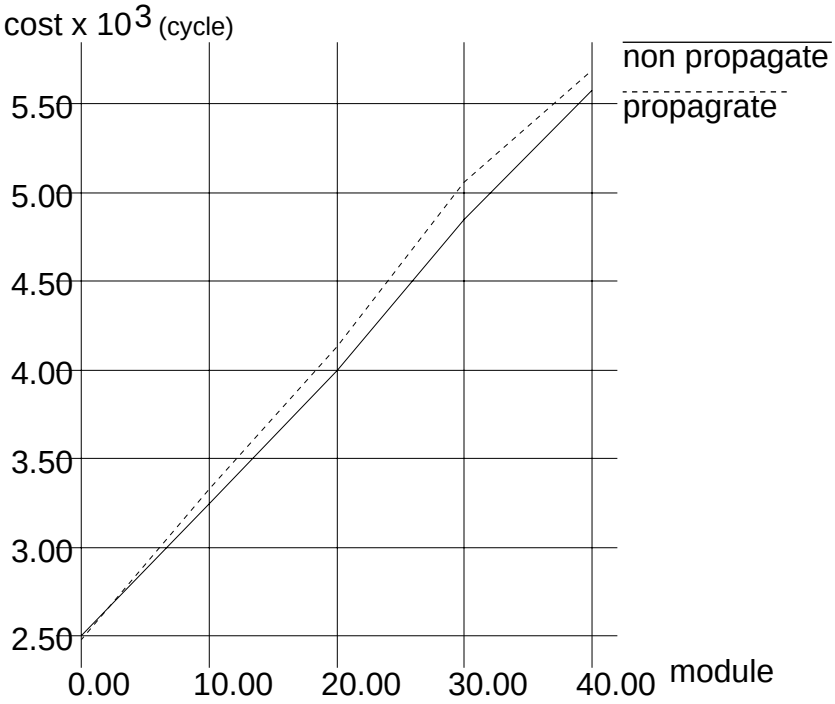


図 6.4: フィルタ伝播の効果 (データ処理なし)

数のみである。この図からはフィルタの伝播による効果は見られない。

次に、同様のフィルタを持ち、データ処理として整数型データの書き込みを一個行うモジュールを合成して、同様に実行サイクルを測定し、結果を図 6.5に示す。このとき、伝播を行わないモジュール集合から生成するコードは、

初期化, 分岐, 書き込み, 分岐, 書き込み, ...

となり、伝播を行う集合から生成するコードは、

初期化, 分岐, 分岐, ..., 分岐, 書き込み, 書き込み, ..., 書き込み

となる。図 6.5より、フィルタの伝播を行った場合のコストの方が小さく、フィルタの伝播に効率を改善する効果があると判断できる。これは、分岐および書き込みを局所化することによりそれぞれのデータ参照の局所化が進み、キャッシュミスが減少したためであると考えられる。

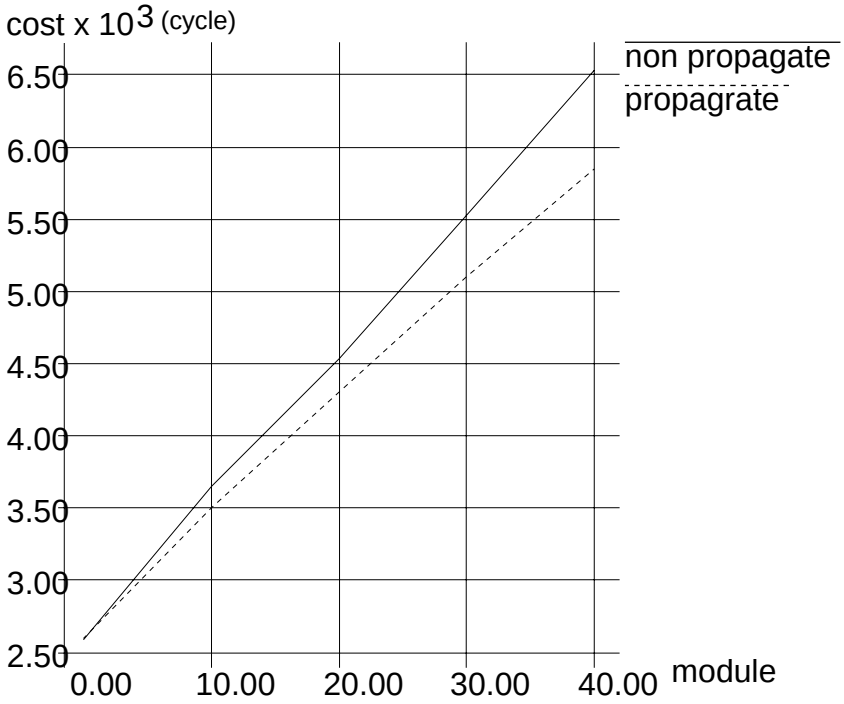


図 6.5: フィルタ伝播の効果 (データ処理あり)

6.4 動的合成モジュール導入のコスト

スペシフィケーションモジュールの生成、モジュールの連結、インプリメンテーションモジュールの生成すなわちスペシフィケーションモジュールの合成にかかるコストを表6.3に示す。スペシフィケーションモジュールの生成はフィルタ無しの場合、インプリメンテーションモジュールの生成はデータ処理コードを生成しない場合のコストを示している。

	コスト (cycle)
スペシフィケーションモジュールの生成	6911
モジュールの連結	3390
インプリメンテーションモジュールの生成	9301

表 6.3: モジュール連結のコスト

通常、表6.3のコスト以外に以下のコストを必要とする。

スペシフィケーションモジュール生成時

- フィルタ生成のコスト
  指定するフィルタの数に対するモジュールの生成サイクルを図6.6に示す。

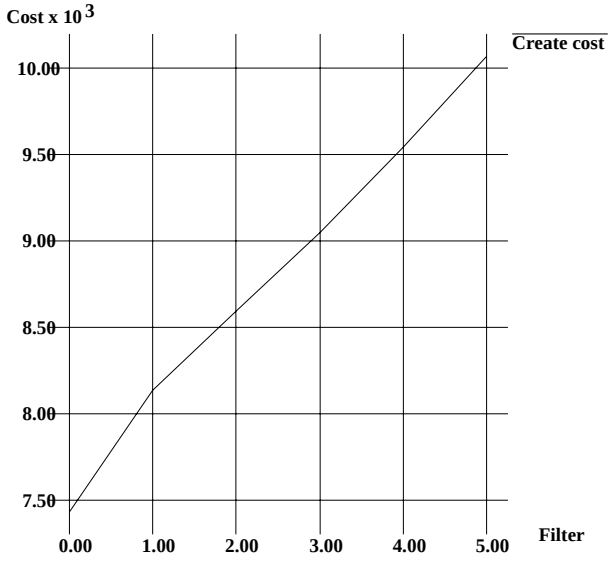


図 6.6: フィルタ数に対するモジュール生成コスト

インプリメンテーションモジュール生成時

- フィルタ伝播のコスト
  現在の実装は、受け取ったフィルタをすべて伝播できるときのみ行い、新たなスペシフィケーションモジュールを生成する。したがって、スペシフィケーションモジュールの生成コストにほぼ等しい。

- データ処理コード生成のコスト
  生成するコード量による。

- 統合するデータ処理コードのためのリスト生成のコスト

将来はデータ処理コード生成関数を動的にカーネルにリンクできるようにするので、そのためのコストも必要となる。

6.5 議論

図 6.3より、一組のデータ処理を統合することによって生成コードの処理を 4〜6 サイクルの短縮することができる。これは、大量のデータ処理を数種類行う場合に役に立つ。例として、ネットワークプロトコルの複数レイヤでのチェックサムの計算やバイトスワップの統合、また映像データや音声データを扱うときのデータ変換を複数回行う場合などが挙げられる。

6.4 節よりモジュールの挿入や合成のコストは比較的大きく、もともとサーバを用いないためにリモートプロシージャコールの削減を見込むことができないアプリケーションや、大量のデータ統合をすることができないアプリケーション、ダウンロードしたモジュールを一度しか使わないようなアプリケーションにはあまり適さない。逆にネットワーク処理のようにパケット到着などのカーネル内イベントによって何度も使用するようなアプリケーションに適している。

以上より本研究の提案に特に適しているアプリケーションをまとめると、

- カーネル内で大量のデータを扱う
- 同一のデータに対して複数回の処理を行う
- 従来サーバで実現されている機能の一部を使用し、その機能はモジュールとして切り出すことができる
- カーネル内イベントによって何度も呼び出される処理を含む

ようなアプリケーションであると考えることができる。

第 7 章

まとめと今後の課題

7.1 まとめ

オペレーティングシステムを拡張してアプリケーションの効率を向上するためにカーネルへダウンロードするコードについて、モジュール化設計を勧める。そのため、モジュール化設計の欠点である実行効率の改善を行う最適化を提案した。提案する最適化は、データ処理の統合によるロード/ストアの削減と、フィルタの導入による制御の流れの最適化を中心としている。提案の有効性を調べるためにプロトタイプの実装を行い、基本的な評価を行った。

7.2 今後の課題

実際のアプリケーションによる評価　提案した最適化の有効性をより明確に示すため、実際のアプリケーションを構築して実行の時間効率について調査する必要がある。対象とするアプリケーションは、モジュール化設計が適しているネットワークプロトコルや、大量のデータの移動を行うビデオサーバ/クライアントなどが挙げられる。

データ処理コード生成関数の動的カーネルロード　現在のプロトタイプ実装では、モジュール化設計の欠点を改善する最適化についての有効性を示すことのみに集中している。そのためデータ処理コード生成関数はカーネルとの動的リンクをしておらず、静的にカーネル内に用意している。本研究での提案が日常的に利用されるためには、任意のデータ処理

コード生成ができることが必要で、そのためにはDKM[追川 96] のようなフレームワークを利用して任意のコード生成関数をカーネルにロードできるようにすることが不可欠である。

謝辞

本研究を進めるにあたって、終始に渡って御指導いただきました中島達夫助教授に心からの感謝を申し上げます。中島研究室の保木本晃弘さんには世の中の研究動向などの興味深いお話をいろいろいただきましたことを感謝致します。中島研究室の赤木敏和さんには Real-time Mach の遊びかたを教えていただきました。ありがとうございました。中島研究室の皆様には研究についていろいろな議論を頂き、ありがとうございました。

参考文献

[AP93] M. B. Abbot and L. L. Peterson. Increasing network throughput by integrationg protocol layers. In *IEEE/ACM Transactions on Networking*, October 1993.

[BGP+94] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safty and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[EK96] D. R. Engler and M. F. Kaashoek. Dpf: fast, flexible message demultiplexing using dynamic code generation. In *ACM Communication Architectures, Protocols, and Applications(SIGCOMM'96)*, August 1996.

[EKJ95] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for apprication-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[Eng96] D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceeeding of the SIGPLAN '96 Conference on Programing Language Design and Inplementation*, May 1996.

[EWK96] D. R. Engler, D. A. Wallach, and M. F. Kaashoek. Design and implementation of a modular, flexible, and fast system for dynamic protocol coomposition. Technical Report TM-552, Massachusetts Institute of Technology Laboratory for Computer Science, May 1996.

[FB96] M. E. Fiuczynski and B. N. Bershad. An extensible protocal architecture for apprication-specific networking. In *Proceedings of the 1996 Winter USENIX Conference*, January 1996.

[LCC94] C.-H. Lee, M. C. Chen, and R.-C. Chang. Hipec: High performance external virtual memory caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation, USENIX Association*, November 1994.

[SESS94] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. An introduction to the architecture of the vino kernel. Technical Report 34-94, Harvard University Center for Research in Computing Technology, 1994.

[WEK96] D. A. Wallach, D. R. Englar, and M. F. Kaashoek. Ashs: Application-specific handlers for high-performance messaging. In *ACM Communication Architectures, Protocols, and Applications(SIGCOMM'96)*, August 1996.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, December 1993.

[追川 96] 追川修一, 杉浦一徳, 西尾信彦, 徳田英幸. 移動計算機環境のための適応的なカーネルオブジェクト管理. 情報処理学会 モーバイルコンピューティング研究グループ研究報告, November 1996.

付録 A

インタフェース

A.1 システムコール

動的モジュール合成(Dymnamic Module Composition)のためのインタフェースを以下に示す。

kern_return_t dmc_module_create(specid, req, type, propagatetype)
dmc_module_create() はカーネル内にスペシフィケーションモジュールを生成し、そのIDを返す。

kern_return_t dmc_manipulate_set(specid, manipid, arg, argsize)
dmc_manipulate_set() はデータ処理コード生成関数をカーネルへロードし、スペシフィケーションモジュールにに関連づける。現在はカーネル内のデータ処理コード生成関数を使っている。

kern_return_t dmc_module_connect(prior, posterior)
dmc_module_connect() はスペシフィケーションモジュールを連結する。

kern_return_t dmc_module_compose(implid, specid, name)
dmc_module_compose() はスペシフィケーションモジュールを動的に合成、コンパイルしてインプリメンテーションモジュールを生成し、そのID を返す。

kern_return_t dmc_module_exec(implid)
dmc_module_exec() はインプリメンテーションモジュールを実行する。

kern_return_t dmc_module_destroy(implid)
dmc_module_destroy() はインプリメンテーションモジュールを削除する。

A.2 データ処理コード生成インタフェース

データ処理コード生成メソッドの記述を容易にするため、vcode へのマクロをいくつか用意した。

dmc_manip_ref_set(specid, addr)
dmc_manip_ref_set() はモジュールが参照しているデータを変更するコードを生成する。

dmc_manip_integrate(specid, offset, size, policy, manip, arg)
dmc_manip_integrate は統合できるデータ処理を統合した後、処理コードを生成する。

dmc_manip_copyout(kern, user, size)
dmc_manip_copyout() はカーネル内データをユーザ空間へコピーするコードを生成する。