# リンク後の実行イメージに対するPortal生成法とその評価

小林 良岳　中山 健　前川 守

## 概 要

実行中のプログラムに対し，その一部を動的に差し替えあるいは拡張するにはプログラムの実行状態を監視する手段が必要である．しかし，状態監視のためにソースコードに変更を加えることは，プログラム作成者の負担となり思わぬミスを招く．我々は，コンパイル時にPortalと呼ばれる状況監視のためのコードを，関数の呼び出しポイントに対して自動生成するPortal Creator(PoC)を実装しているが，生成の方針が静的であり，一旦Portalを生成したコードは必要がない場合でも常にPortalを通過するため処理時間に対するPortalのオーバーヘッドが加わるという欠点がある．そこで本稿では，コンパイル時にはプログラムをPortal生成に十分となるように修正するにとどめ，Portalの生成を実行時や実行中に延期できる方法を提案する．また，評価によりPortalを生成するために必要な情報を組み込んだ実行イメージのパフォーマンスが実用の範囲内であることと，実行後にPortalを組み込んだ場合のパフォーマンスが従来のPortalと同程度であることを示す.

# Run-time Portal Creation and Its Evaluation

Yoshitake KOBAYASHI　Ken NAKAYAMA　Mamoru MAEKAWA

## Abstract

In order to dynamically replace and extend programs while they are running, some mechanisms to monitor the current status of them is necessary. However, regarding modifications to the programs for monitoring would be a burden to the programmers and error-prone. We already proposed Portal Creator(PoC) which automatically generates a Portal for each function at compile time for the above purpose. But once Portals are created, all function calls must go through Portal even if it is not used. In this paper, we propose a method for “lazy” Portal creation, can be postponed until execution time or run-time. At compile time, PoC just modifies program structure and creates informations to be needed for portal creation. We evaluate the performance of this method on program execution.

## 1　はじめに

システムが動的に変更可能であれば，システム上で動作中のプログラムにセキュリティパッチの適用などの必要性が生じた場合でも，動作しているプログラムに対し機能の追加／削除や変更を行うことで対応できる．これは，システム停止に伴うコスト [1] の削減にもつながる．しかし，動作中のプログラムはそれぞれ部品が何らかの状態を持っているため，プログラムの詳細な実行状態を監視し，差し替え直前までの状態を保持したまま変更可能なシステムを提供する必要がある．

実行状態を監視するシステムの実現法としては，モニタするための専用のシステムコールを作成しているもの [2]，リフレクションを用いるもの [3] などがある．しかし，どの場合もプログラムに対してプログラマが変更を加えねばならず，その作業

電気通信大学 大学院情報システム学研究科 情報システム設計学専攻
Department of Information Systems Science, Graduate School of Information Systems, University of Electro-Communications

量は無視できない．また，処理時間に対するオーバヘッドも問題となる．

我々は，プログラム部品の実行状態を判別するための Portal と呼ばれる機構を，プログラムのコンパイル時に，全てのプログラム部品に対して自動的に付加することを提案している [4]．Portal を用いることの利点は，プログラマに対してソースコードの変更を要求しないこと，また状態監視を必要としない時の状態保持を，カーネルなどの外部のプログラムに頼らず，個々のプログラム内部で行なうことによりオーバヘッドを抑えているということ，および実行状態の監視が必要なときだけ，カーネルに監視させることが可能であるということにある．

しかし，この手法はプログラムコンパイル時に必ず状態保持のために Portal を埋め込む必要があるという欠点がある．つまり，一旦 Portal を生成してしまうと，多少のオーバーヘッドが生じることは避けられない．プログラムの種類によっては，同一の名前をもつプログラムを，システム上に常駐するデーモンプロセスとして実行したり，常駐しないプロセスとして実行したりすることがある．常駐しないプロセスに対して Portal を付加することは，単にオーバーヘッドを増加させるだけであまり意味がない．

そこで，本稿ではリンク後のプログラムに対して，プログラム実行時や実行中に Portal を埋め込む手法を提案する．この手法により，普段はあまり Portal の必要性がないプログラムが，処理のパフォーマンスをほとんど犠牲にしなくても良くなるだけでなく，後で必要に応じて Portal を付加することで再構成を行うことも可能となる．

以降の議論では，実装対象となるプログラム言語を C 言語とし，単に関数といった場合は C 言語の関数を指す．またモジュールとは，1 つのソースファイルをコンパイルしてできるオブジェクトファイルを指し，これらモジュールをリンクすることにより実行可能イメージができる．そして，実行可能イメージを OS 上で実行する際の単位をタスクとする．モジュール差し替えの最小単位は，モジュール内部の関数である．

## 2　従来の Portal の実装

### 2.1　Portal の構造

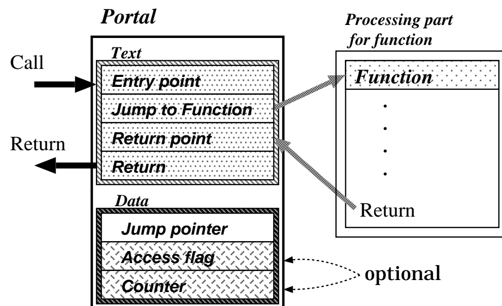


図 1: Portal の構造

Portal は図 1 に示す構造を持つ．Portal の基本的な役割は，関数の処理に対する呼び出しポイントと，関数の処理からの復帰ポイントを提供することである．これを利用することで関数の使用状況を調べたり，関数呼び出しの制御を行うことが可能となる．

Portal には，基本データとしてジャンプポインタがある．ジャンプポインタは，関数の処理部分を呼び出す際に用いられる．つまり Portal から処理部分の呼び出しはポインタを用いたジャンプとなる．そして，このジャンプポインタの値を変更することにより関数の処理部分を切り替えることが可能である．それ以外のデータについては，オプションとして提供する．関数の呼び出し状態把握を行なう Portal では，モジュール内の関数の利用状況を把握するために，カウンタ，アクセスフラグという 2 つのデータが追加される．以降の議論では，状態把握用の Portal に焦点を当てる．

カウンタは，関数を呼び出す際に 1 インクリメントされ，関数の処理が終了しリターンする前に 1 デクリメントされる．カウンタの値が 0 であるということは，その Portal によって管理されている関数の処理部分が，その時点で使用されていないことを示す．もし，関数の呼び出し回数の統計を取りたい場合は，Portal 生成時にカウンタをデクリメントしないものを生成すれば良い．

アクセスフラグは，Portal に対して呼び出しがあると 1 にセットされる．これを利用して，モジュール差し替え時に状態再現を実行または再実行する必要があるか否かを判断できる．アクセスフラグが 1 であることは，すでにモジュールが一度は利用されたことを示し，状態再現を行なう必要がある．また，状態を再現する前にアクセスフラグを 0 にしておき，状態再現が終了した時にもう一度ア

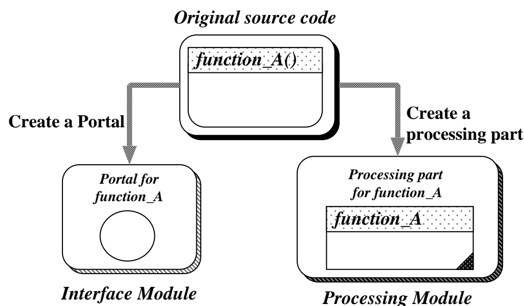


図 2: 2 つに分割されるモジュール

クセスフラグを参照する．この時アクセスフラグが 1 にセットされているということは，状態再現中にモジュールに対して何らかのアクセスが行なわれたことを示し，状態再現を再実行する必要があると判断できる．

### 2.2 インタフェイスと処理の分離

通常は 1 つのソースファイルから 1 つのモジュールが生成されるが，Portal の導入により 2 つのモジュールが生成されるようになる (図 2)．1 つは Portal のモジュールで，これを**インタフェイスモジュール**と呼ぶ．もう 1 つは，関数の処理部分のモジュールで，これを**処理モジュール**と呼ぶ．Portal と関数の処理部分は 1 対 1 に対応するので，インタフェイスモジュールと処理モジュールも 1 対 1 に対応する．

### 2.3 従来の Portal の問題点

今までの実装では Portal はコンパイル時に自動生成することが前提であった．例えば，C 言語で書かれた内部で何もしない関数 `func1()` を IA-32[5] 用のアセンブラコードに直すと，図 3 のようになる[1]．これに対して，インタフェイスモジュールと処理モジュールを生成すると図 4 と図 5 に示すコードが作成される．このコードでは Portal から処理モジュール内部の関数を呼び出す際にも，処理モジュールから Portal への復帰にも `jmp` 命令を用いていた．

Portal を生成しないでコンパイルされリンクされる実行イメージは，通常 `call` 命令で関数呼出しが行なわれ，呼ばれた関数は `ret` で戻る．つまり，Portal の生成は，今までの関数呼出しの構造を変更してしまっていたという事になる．また，2 つの

```
 1: .text
 2: func1:
 3:     pushl %ebp
 4:     movl  %esp,%ebp
 5:     leave
 6:     ret
```

図 3: func1() のアセンブラコード

```
 1: .data
 2: _pcounter_func1:  .long   0
 3: _aflag_func1:     .long   0
 4: _jpointer_func1:
 5:          .long   entp_func1
 6:
 7: .text
 8: func1:
 9:     jmp     1f
10: 1:  incl    _pcounter_func1
11:     movl    $1, _aflag_func1
12:     jmp     *_jpointer_func1
13: rpoint_func1:
14:     decl    _pcounter_func1
15:     jmp     2f
16: 2:  ret
```

図 4: func1() に対する Portal

```
 1: .text
 2: entp_func1:
 3:     pushl   %ebp
 4:     movl    %esp,%ebp
 5:     leave
 6:     jmp     rpoint_func1
```

図 5: func1() の処理部

モジュールを生成していることで，リンクの作業が複雑になるという欠点もあった．

## 3 パフォーマンスに関する考察

Portal を提案した当初の目的は，関数呼び出しの状態把握のためであった．そして，Portal を生成する際のアプローチを決めるにあたり，処理に対するオーバーヘッドが問題となり，これを極力抑える必要があった．そのために，これまでの実験[2]をもとに Portal の実装による処理のオーバーヘッドを評価し直す．

まず，Portal をそれぞれ C 言語の関数とアセンブラを用いて実装し，Portal を介して何もしない

[1] 以降，アセンブラコードは全て IA-32 を対象とする．

[2] 実験は，Intel PentiumIII 500MHz，メモリ 256MB を搭載した PC/AT 互換機上に FreeBSD-3.4 をインストールして行なったものである．

表 1: 1 回の Portal 処理時間

| Portal 実装言語 | 処理時間 $[ns]$ | 比 |
|---|---|---|
| アセンブラ ($P_a$) | 38.1 | 1 |
| C 言語 ($P_c$) | 70.4 | 1.84 |

表 2: 関数呼び出しとシステムコールの処理時間

| | 実行時間 $[ns]$ | 比 |
|---|---|---|
| 関数呼び出し ($C_f$) | 21.8 | 1 |
| システムコール ($C_s$) | 826.1 | 37.9 |

表 3: Emacs を make するために必要なシステム CPU 時間

| Portal 生成の方針 | 時間 $[s]$ | 比 |
|---|---|---|
| Portal なし ($E_n$) | 4.14 | 1 |
| public 関数のみ生成 ($E_p$) | 4.99 | 1.21 |
| 全ての関数に生成 ($E_a$) | 5.23 | 1.26 |

関数 (図 3) を呼び出した時，表 1 に示す差が生じた．表 1 の結果では，C 言語で作成した Portal はアセンブラで作成したものと比べて処理時間が約 1.8 倍になることが示されている．この差が生じる原因としては，C 言語で作成した場合 Portal を呼び出すためにスタックを操作し，もう一度処理部分を呼び出すためにスタックの操作を行なうということが考えられる．

さらに，C 言語の関数呼び出しとシステムコールの差を比較することにより (表 2)，システムコールを利用して実装するというアプローチは一般の運用において，効率面に関しては有利でないと考えた．ただし，モジュール差し替え時にはカーネルレベルでプログラムの実行状態を確保する必要があり，その時だけはオーバヘッドを伴ったとしても，カーネルに対して作業を依頼することは必要である．これを実現するために，Portal には必要に応じてカーネルに制御を移行できるようになっている [4]．

また，FreeBSD のカーネルに対し Portal の生成を行い，Emacs-20.5 の make を行なった時のシステム CPU 使用時間を測定した (表 3)．ユーザ CPU 時間については，どの場合も約 67.7 秒になった．

表 1 から表 3 の結果を利用することにより，状態把握をシステムコールで実装した場合と，Portal を C 言語で実装した場合のオーバーヘッドを求めることができる．

まず，次式によって，それぞれのアプローチにおけるシステム処理時間 $T$ が求められる．

$$T = \frac{E_a - E_n}{P_a - C_f} \times \alpha + E_n \quad (1)$$

ここで $\alpha$ は，システムコールを利用する場合 $C_s \times 2$，C 言語で Portal を実装する場合 $P_c - C_f$ となる．システムコールの場合に値を 2 倍しているのは，関数呼び出しの前後でシステムコールを実行する必要があるからである．

そして，式 1 に実際に値を入れて時間を求めると状態把握をシステムコールで行った場合は 114.62 秒，C 言語で Portal を実装した場合は 7.38 秒となった．この結果をそれぞれ処理時間に対するオーバーヘッドとして計算すると，2668%，78%となる．ただし，この結果はあくまでも概算であり，実測した場合はパイプライン処理などの影響により，小さくなるとも考えられる．

しかし，何らかの処理をプログラムに加えたとき，少なからず全体の処理への影響があることは明確である．そこで，オーバーヘッドを極力抑える必要性がある．

## 4 リンク後の Portal 生成手法

本章では，リンクされた実行イメージに対して Portal を生成するための手法を提案する．基本的な考え方は，以下の 2 つである．

- 関数呼び出し形式を直接呼び出しから間接呼び出しへ変更．
- 呼び出しからの復帰の際，Portal を通過可能なプログラム構造へ変更．具体的には，`ret` 命令に対して位置の一覧を作成し，`ret` 命令を Portal への `jmp` 命令に書き換え可能な構造にする．

### 4.1 関数呼び出しの視点

Portal の生成について述べる前に，まず関数呼び出しの視点について確認する．関数の呼び出しは，呼び出し側から見る場合 (図 6) と，呼び出される側から見る場合 (図 7) がある．どちらの視点から考えるかによって，リンク後の Portal 生成に関する方針が変わる．

Portal が生成されていないコンパイル後のモジュールがリンクされ，メモリ上で実行されている

時，関数呼出しは call 命令で行なわれ，呼び出された関数から呼び出した関数へは ret 命令で戻るということが前提となる[3]．このリンク後のイメージに対して Portal を生成することを考えた時，次の 2 点を考慮する必要がある．

- 関数呼び出し時に Portal を通過させる方法
- 呼び出された関数から Portal を経由して復帰する方法

### 4.2 関数呼び出し時の Portal 通過手法

まず，関数の呼び出し方法についてであるが，Portal を生成した後に関数呼び出しをする時，Portal を呼び出せるようにするためには，呼出し先を関数本体ではなく，Portal にリダイレクトする必要がある．この時，次に示す 2 つの方法が考えられる．

- Portal を生成する必要がある関数を呼び出しているポイントを全て見つけリストを生成する
- コンパイル時やソースの作成時に全ての呼び出しポイントを見つけて関数毎にリストを生成しておく

まず，Portal を生成する必要がある関数を呼び出しているポイントを全て見つける手法であるが，これは実行中のイメージの全ての部分に対してコードの検証を行なう必要がある．これは，その処理の複雑さの面から見たとき，適当ではない．

次に，呼び出しポイントのリストをコンパイル時やソースコード作成時に作成する手法であるが，1 つの関数呼ばれる場所がいくつあるかについて考

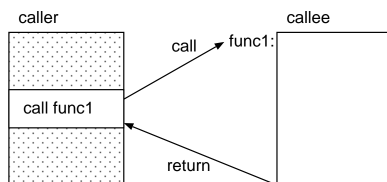


図 6: 呼出し側からみた関数呼出し

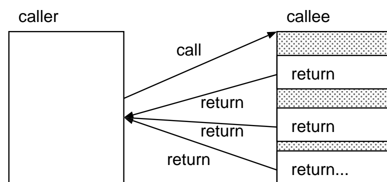


図 7: 呼ばれる側からみた関数呼出し

[3]大域脱出のケースを除いた場合

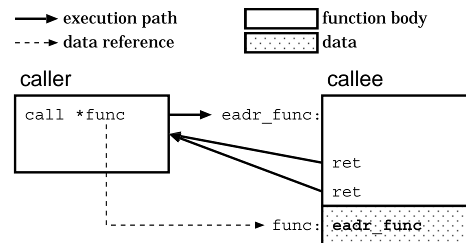


図 8: ポインタを利用した間接呼び出し

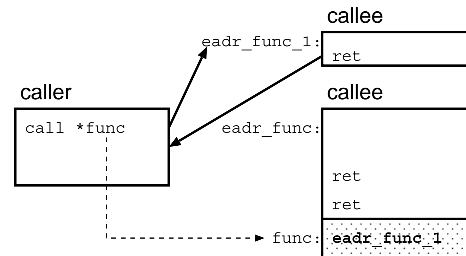


図 9: ポインタ書き換えによる呼び出し先変更

えた時，それはプログラミングに依存することとなる．つまり，関数呼び出しの関係が多ければ多いほど，始めに生成しておかなければならないリストが多くなり，リソースの面から有効ではない．

これら 2 つの手法では，管理しなければならないものが，対象関数の呼び出し部分であることが前提であった．そして，この 2 つの手法に共通する欠点としてあげられるものは，管理しなければならない呼び出し部分が分散しているという点である．これは，図 6 の視点で見ているからである．

もし管理しなければならない呼び出しポイントが 1 つになれば，この問題は解決される．そこで，関数呼び出しを図 7 の視点から考え，呼び出される側が正しい呼び出しポイントに導くという手法をとる．これは，従来の Portal で用いていた手法と同様に，ポインタを用いた間接呼び出しにすることで実現できる (図 8)．そして，Portal を生成する際には，関数を指すポインタを変更することにより Portal や任意の関数へのリダイレクトを行なう (図 9)．

### 4.3 関数から Portal を経由する復帰手法

Portal を動的に追加する場合，呼び出された関数から復帰する際にも Portal を通過する手段を提供する必要がある．

関数を呼び出す側から見た時 (図 6)，呼び出す関数は呼び出した関数から ret 命令で復帰するこ

とを期待する．つまり，呼び出された関数のどの部分から戻ってくるということは問題ではない．呼び出し時には，復帰ポイントがスタックにpushされ，呼び出された関数から復帰する際にはpopされる．この状態でPortalを生成することを考えたとき，呼び出し時にPortalを呼び出せたとしても，スタック上にpushされている復帰ポイントは呼び出した関数の呼び出しポイントの次の命令を指しており，このままでは復帰時にPortalを通過することができない．

そこで，この問題を解決するために，復帰ポインタを一旦保存し，書き換えてから対象関数を呼び出すという手法が考えられている [6]．しかし，この手法では，復帰ポインタを保存するためのコストがかかる．これは，今までのPortalの実装と比較した場合，Portalを生成した後のオーバヘッドが大きくなり，パフォーマンスへの影響が大きい．

既に [4] では，関数の呼び出し状態を把握するためのPortalのオーバヘッドが検証されているが，その実験でもおよそ20%から30%のオーバーヘッドがかかっていた．よって，復帰ポインタを保存するためのコストをさらに加えることは，好ましくないと考えられる．復帰ポインタ保存のための処理時間に対するコストは，次の2つの作業によるものである．

- 復帰ポインタの保存するための領域指定
- Portalへ戻るための復帰ポインタの再設定

これを実現するためのは，復帰ポインタを保存するための領域を確保する必要がある．パフォーマンス面で有利になるように領域を確保するには，連続領域を確保しておくと良い．もし，保存領域が分散した場合，そのリストをハッシュなどで確保しておく必要があり，リスト操作のためのコストが余計にかかるからである．

しかし，復帰ポインタ保存のための連続領域を確保した場合においても問題が発生する．例えば，再帰呼び出しが頻繁に行なわれるプログラムでは，復帰ポインタを保存する領域が増え続け，十分な領域が確保されていなかった場合に，プログラムが正しく実行されなくなる．

そこで，図7に示した呼び出された関数からの視点で考えると，関数呼び出し時に復帰ポインタを保存しないアプローチが考えられる．つまり，コンパイル時に処理モジュールに対してPortalへの復帰をすることを予想したコードの生成を行なうことで復帰するポイントを把握し，そこを書き換えることによりPortalに対して制御を渡すことが可能となる．これにより，リスト操作やスタック操作に関する処理が省略され，パフォーマンス面では有利となる．この変更に関する詳細は，次章で述べる．

## 5 Portal Creatorへの変更

### 5.1 PoCの機能と変更点

今までの議論をもとに，PoCへの変更を行なった．PoCはPortalをコンパイル時に自動生成するプログラムである．これにより，プログラマが手作業でプログラムへ変更を行なう場合と比べ，Portalを生成するためのプログラム作成時のミスが軽減される．従来のPoCの機能には以下のものがある．

1. Portalの生成
2. モジュール内部のシンボル情報の生成
3. モジュール内部から参照される関数やデータへのポインタの生成

今回の変更点は，主に1に関するものであり，その変更は次に示すものである．

- Portalを直接生成せず，後にPortal生成に必要となる情報を生成
- インタフェイスモジュールと処理モジュールという2つのモジュールを生成せず，処理モジュールのみを生成

### 5.2 Portal生成用の情報の生成

今までのPoCでは，どの関数にPortalを生成するかに関して，以下の2つの方針をコンパイル時に選択できた．

- 全ての関数：モジュール内部でのみ使用されるprivate関数を含む全ての関数に対して生成．
- Public関数：モジュールの外部に公開しているpublic関数に対してのみ生成．

Portalの生成をリンク後に行なうことが可能になることにより，上記の生成方針も実行中の実行イメージに対してPortalを生成する際に指定すればよい．そこで，PoCでは全ての関数に対してPortalが生成可能となるようにしておく必要がある．

PoCによる情報の生成は，次の手順で行なわれる．

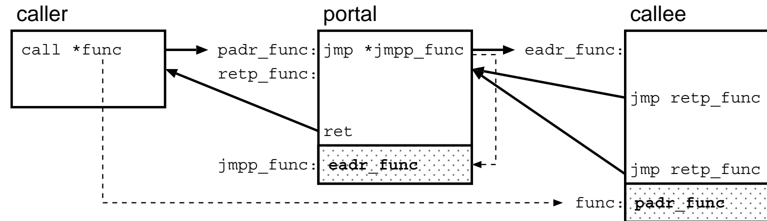


図 12: Portal 生成後の状態

1. ソースコードをコンパイルする時，直接オブジェクトを生成せずに，アセンブラコードを生成
2. 生成されたアセンブラコードをもとに関数のエントリとシンボル情報を全てとりだす
3. 全ての関数のエントリポイントを `call` が使うポインタに変換
4. `call` 命令をポインタを用いた間接呼び出しに変換
5. 全ての `ret` 命令にタグを付ける
6. `ret` 命令の後ろに復帰ポイントへのポインタ格納スペースを確保
7. 最後に生成されたアセンブラコードをアセンブルして処理モジュールを生成

例えば，図 10 に示す C 言語のコードから PoC によって生成されるアセンブラコードは図 11 のようになる．

```
 1: int func1(){func2();}
 2: int func2(){}
```

図 10: オリジナルの C 言語のコード

```
 1: .text
 2: entp_func1:
 3:     pushl %ebp
 4:     movl  %esp,%ebp
 5:     subl  $8,%esp
 6:     call  *func2
 7:     leave
 8: tagret_func1_1:
 9:     ret; nop; nop; nop; nop
10: entp_func2:
11:     pushl %ebp
12:     movl  %esp,%ebp
13:     leave
14: tagret_func2_1:
15:     ret; nop; nop; nop; nop
16:
17: .data
18: func1:   .long entp_func1
19: func2:   .long entp_func2
```

図 11: PoC によって変換されたアセンブラコード

## 6 Portal の生成

Portal を利用する必要があるタスクに対しては，実行時または実行後に Portal の生成を行うことが可能である．まず，PoC によって生成され実行されるコードは，図 8 に示した構造を持つ．図 8 の `callee` に対する Portal 生成を例にとると，Portal の生成は次の手順で行われる．

1. Portal を生成する関数用の Portal のコードを生成
2. 関数内でタグを付けられている `ret` 命令の後ろに確保されている復帰ポインタ領域に，Portal 内の復帰ポイントへのアドレスを設定
3. 関数へのポインタを Portal に向ける
4. 関数内の `ret` 命令を Portal に対する `jmp` 命令に書き換える

この作業により，図 8 に示した関数呼び出しと復帰は，図 12 のように変更される．

## 7 評価

PoC による関数呼び出し形態変更による処理時間への影響を調べるために，評価を行なった．評価には図 13 に示す構造をもつプログラムを用いた．メインループから関数を 1 億回呼び出し，1 回の平均値を求めた．実験は，PentiumIII 550MHz，メモリ 256MB を搭載した PC/AT 互換機に Aya オペレーティングシステムをインストールして行った．結果を表 4 に示す．

表 4 の結果では，PoC による関数呼び出しの変更を行ったものには，何も変更を行わなかったものと比べて，約 11%のオーバーヘッドが生じていることがわかる．これは，呼び出し形式を直接呼び出しから間接呼び出しに変更した影響である．

また，従来の Portal と，PoC による変更を行った後に Portal を生成したものとの間には，変更な

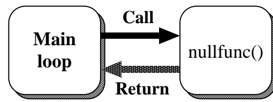


図 13: 評価に用いたプログラムの構造

表 4: コード変更による処理時間への影響

| コードへの変更 | 時間 ($ns$) | 比 |
|---|---|---|
| 変更無し | 20.7 | 1 |
| 従来の Portal | 34.9 | 1.68 |
| PoC による変更のみ | 23.0 | 1.11 |
| PoC+ Portal 生成 | 36.8 | 1.77 |

しのものを基準にした場合に約 9%のオーバーヘッドの増加となり，従来の Portal を基準にした場合は，約 5%のオーバーヘッドの増加となった．

しかし，以上の実験は，関数内で何も処理を行わなかった場合を対象としたものである．表 4 の結果より，変更を行わないものと PoC による変更を行ったものの差を求め，その差を 500MHz の CPU を利用した場合に換算してから式 1 を利用してオーバーヘッドを求めると約 4%となる．また，全ての関数に Portal を生成した場合は約 29%となり，表 3 の結果と比べると約 3%の増加にとどまる．よって，これらの結果から，PoC による変更の処理への影響は少ないと考えられる．

## 8 関連研究

EEL[7][6] や MDL[8] では，実行中のプログラムに対して変更するための手段を提供している．しかし，新しく挿入するコードを実行するために，レジスタの保存が必要であったり，実行中のコードの解析が必要である．これらのアプローチは，本稿で示したコンパイル時に情報を生成しておくというアプローチと比べ，処理のパフォーマンス面で不利である．

また，文献 [9] や結 [10] 動的リンカでは，リンク機構を利用して実行状態把握をサポートしている．しかし，これらの手法では動的リンカを利用する特性上，静的リンクされたコードに対するコードの生成や状態把握ができないという欠点がある．そこで，本稿で示した手法と，組み合わせて利用することにより，柔軟にプログラムの構成を変更したり実行状態把握を行うことが可能となると考えられる．

## 9 まとめ

本稿では，リンク後の実行イメージに対して実行時または実行後に Portal を生成する手法を示した．この手法では，モジュールを作成する際に PoC によってあらかじめ Portal を生成するための機構を組み込んでおくことにより，Portal 生成の要求に柔軟に対応することが可能である．また，この変更による処理パフォーマンスへの影響は少なく，実用に耐える程度であることが示された．今後は，リンカへの変更を行うことで，静的リンクと動的リンク両方の手法が利用可能となる方式を提供する．

## 参考文献


[1] David A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of LISA 2002 Sixteenth Systems Adinistration*, pp. 185–188, Berkeley, CA, 2002.

[2] 谷口秀夫, 伊藤健一, 牛島和夫. プロセス走行時におけるプログラムの部分入替え法. 電子情報通信学会論文誌, Vol. J78-D-I, No. 5, pp. 492–499, May 1995.

[3] 森大志, 前川守. 実行時の状態再現を伴うソフトウェアモジュールの動的差し替え機構. 第 55 回 (平成 9 年度後期) 全国大会講演論文集 (1), pp. 224–225. 情報処理学会, September 1997.

[4] 小林良岳, 佐藤友隆, 唐野雅樹, 結城理憲, 前川守. 彩：コンパイル時に自動生成される Portal をもとに動的再構成可能なオペレーティングシステム. 電子情報通信学会, Vol. J84-D-I No.6, pp. 605–616, June 2001.

[5] *The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 2000.

[6] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report CS-TR-92-1083, Madison, WI, USA, March 1992.

[7] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291–300, 1995.

[8] Jeffrey K. Hollingsworth, Barton P. Miller, M. J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of IEEE 1997 Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, pp. 201–213, 1997.

[9] 山本淳, 谷口秀夫. 動的リンクを利用した実行中プログラムの部分入替え法における状態把握法. 情報処理学会研究報告, 2002-OS-91, pp. 141–149, 2002.

[10] 小林良岳, 唐野雅樹, 結城理憲, 紅谷順, 姜亨明, 中山健, 前川守. モジュール差し替え時のプログラム構造変化に対応する動的リンカ. コンピュータシステム・シンポジウム論文集, pp. 65–72, November 2001.