# プロファイルを利用した統合的コード最適化システム

2D−8

中村 さおり　城倉 梨香　三浦 敬至

日本電気(株) ULSI システム開発研究所

## 1 はじめに

アプリケーション・プログラムが大規模化，複雑化するにつれ，従来はコード・サイズが小さいことが優先とされていた組み込み系のシステムにおいても，ターゲット・チップの性能を最大限に引き出した高速な処理が求められてきている．筆者らは，組み込み系アプリケーションのプログラマが，プログラムを実行して得られたプロファイルを利用して最適化を行なうプロセスを容易に実行できるような，チューニング・システムを開発中である．その第一段階として，関数呼び出し情報を利用したキャッシュ最適化を実装，評価した[1].

ところで，実際のアプリケーション・プログラムにおいては，特定の関数，あるいは関数内の特定の箇所が頻繁に実行されるというプログラム固有の特徴がしばしば観測される．こうした箇所を重点的に最適化することにより，さらなる高速化が期待できる．そこで筆者らは，本システムの第二段階として，関数内のプロファイル（各パスの実行回数）を利用した最適化機能を実装，評価したので，本稿ではそれに関して述べる．

## 2 システム概要

本システムの主要な機能は[1]で示したように，プログラムの性能（実行時間，キャッシュ利用状況）を評価する「性能評価部」，関数間プロファイル（関数呼び出しの実行情報）を利用してキャッシュ最適化（関数の再配置）を行なう「プロファイル - キャッシュ最適化部」，GUI を用いて関数の配置を変更可能とする「ビジュアル・リンカ」で，これに加えて「関数内プロファイル最適化部」が今回新規に追加された．ユーザ（プログラマ）は GUI を持つ「ドライバ」を通してこれらの機能を利用する．ドライバはプロジェクト管理機能等も備えている．

本システムの処理フローを図1に示す．

A Code Performance Tuning System using Inter- and Intra-procedural Profile.
Saori Nakamura, Rika Joukura, Takayuki Miura
ULSI System Development Laboratories, NEC Corporation

まず，対象プログラムのオリジナル実行モジュールを生成して性能を評価する．次に，関数間プロファイリング（関数呼び出し情報の収集）を実行する．このプロファイルは後のキャッシュ最適化で利用するものであるが，関数の実行回数の情報が得られるため，これを基に頻繁に実行される関数を判別することができる．

次に関数内プロファイル最適化処理を行なう．対象とする関数をユーザが GUI を用いて指定し，それらの関数に対して関数内プロファイリングを実行し，得られた関数内プロファイルを利用して各種最適化を行なう．この後再び性能を評価し，必要に応じて対象関数を追加してこの処理を繰り返す．

関数内プロファイル最適化の終了後に，キャッシュ最適化を適用する．これは，関数内プロファイル最適化で関数のサイズが変更されることによりキャッシュ最適化へ影響が出るためである．最後に再度性能を評価し，必要に応じてビジュアル・リンカを用いて手動で関数の配置の微調整を行なう．

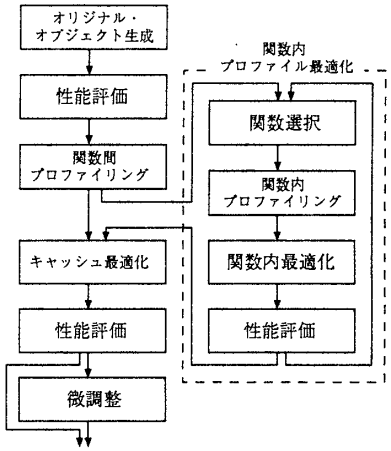


図1: 処理フロー

## 3 関数内プロファイル最適化

### 3.1 プロファイリング

関数内プロファイリングは，パス・プロファイリング手法[2]をベースとしたアルゴリズムによりプローブ・コードを挿入したプロファイリング用実行モジュールを実行し，結果をメモリへ書き出す方法を用いている．

この結果, 関数の入口と出口の間で通過した (実行された) パスとその各々の実行回数の情報が収集される.

### 3.2 最適化

本システムは現在, 以下の 3 種類のコード変形を実装している.

1. スーパーブロック [3] の生成
2. 基本ブロックの並べ替え
3. 命令スケジューリング

まず, 関数内で頻繁に実行されるパス (以下, ホット・パスとする) の途中に他のパスからの合流があった場合, それ以降の基本ブロックを複製し他のパスは複製された方へ合流するように変更することで, 分岐を含まない命令列 (スーパーブロック) を生成する. これは, ホット・パス内で直列部分を拡大することで後の最適化の機会を増加させるための変形である.

次の基本ブロックの並び替えアルゴリズムとしては, ボトムアップ手法やトップダウン手法 [4] 等が知られているが, 本システムの評価ターゲットとして今回用いた V830 (NEC 製 32 ビット RISC チップ) は条件分岐命令で分岐可能な距離が比較的短い, という特徴を持っている. この場合, 基本ブロックの並び替えにより条件分岐先の基本ブロックが遠くへ離れると, 一命令では分岐できず二命令必要となってしまうことがあり, このため一般的なアルゴリズムは適用できない. 従って本システムでは, if-else 構造等で頻繁に実行される方を合流の直前に配置することで無条件分岐回数を削減し, この際にプロファイルに加えて基本ブロックのサイズの情報を考慮する, という手法をとっている.

さらに, スーパーブロック化されたホット・パスに対して, 基本ブロックを越えたスーパーブロック内命令スケジューリングを適用して, 効果的にパイプラインのハザード回避を図る.

## 4 評価

本システムの評価環境は, ターゲットとして V830, ホストとしてワークステーション (Solaris) , プログラムを実行するエンジンとして Solaris 上で動作する命令シミュレータを用いた. この命令シミュレータは, 実行時間の他, パイプライン情報やキャッシュ利用状況等の情報を得ることができる.

サンプル・プログラムとして, ファクシミリ向け圧縮伸長ミドルウェアのエンコード処理を用いた結果を以下に示す. オリジナルの実行モジュールにおいて, コンパイラによる静的な最適化は適用済みである.

このプログラムでは, 全 11 個の関数のうち実行頻度の高い 3 個の関数に対して関数内プロファイル最適化を適用した.

ここでは, 実行頻度が 100 回以上でかつより頻度の高いパスと交わらないものを “ホット・パス” と定義したが, 潜在的な (理論的に可能な) パスのうち実際に実行されたのは約 5% にすぎず, ホット・パスとして認識されたのは各関数で 1 個ずつであった.

表 1 に最適化の結果を示す. 最初が関数内プロファイル最適化のみを適用した場合, 次がキャッシュ最適化のみを適用した場合, 最後が両方を併用した場合の結果で, オリジナルを 100% とした場合の実行時間の比である.

表 1: 評価結果

| 最適化 | 実行時間比 |
|---|---|
| 関数内 | |
| (基本ブロックの並べ替えのみ) | 96.5% |
| (全て) | 94.6& |
| キャッシュ | 99.9% |
| 関数内 & キャッシュ | 94.4% |

本プログラムはキャッシュ最適化が効きにくいケースであったが, 関数内プロファイル最適化を追加することにより, コンパイラの通常の最適化が適用済みの場合であっても, さらに 5% 以上の性能改善が得られることが確認された.

## 5 まとめ

本稿では, 従来のキャッシュ最適化に加えて, 頻繁に実行される関数に対して関数内プロファイル最適化を適用することでさらに性能向上を図るシステムを構築し評価した結果を示し, その有効性について報告した.

今後は, 関数内プロファイル最適化項目の充実を図り, ホット・パスからの冗長なコードの移動等の最適化も実装する予定である.

## 参考文献

[1] 船間, 磯崎, “プロファイルを利用したコード最適化システム”, 情報処理学会第 56 回全国大会, 1998

[2] T. Ball, J. R. Larus, “Efficient Path Profiling”, Proc. of the 29th Annu. Int. Sym. on Microarchitecture (MICRO-29) , pp.46-57, 1996

[3] P. P. Chang, S. A. Mahlke, W. W. Hwu, “Using Profile Information to Assist Classic Compiler Code Optimizations”, Software Practice and Experience, Dec. 1991, Vol.21, No.12

[4] K. Pettis, R. C. Hansen, “Profile Guided Code Positioning” , Proc. of the ACM SIGPLAN'90 Conf. on Prog. Lang. Design and Impl. (PLDI), pp.16-27, 1990