計算機アーキテクチャ 132－25
システムソフトウェアとオペレーティング・システム 80－25
ハイパフォーマンスコンピューティング 75－25
（1999．3．5）

コード再配置による命令キャッシュミスの削減

橋本 敬介† 安藤 秀樹† 島田 俊夫†

近年のプロセッサでは，命令キャッシュを有効利用することが非常に重要となっている。本論文では，命令キャッシュミスを削減するためのコード再配置手法を提案する。本手法を用いることにより，8K バイト の命令キャッシュの場合，SPECint95 ベンチマークの命令キャッシュミス率が m88ksim で 4.14% から 0.00% に，perl で 3.81% から 1.79% に減少した。この減少率は，プロセッサの性能を著しく改善するだけの量であり，4 命令発行のスーパースカラプロセッサの性能を m88ksim で 1.61 倍，perl で 1.23 倍に向上させることができる。

Instruction Cache Miss Reduction by Code Placement

KEISUKE HASHIMOTO ,† HIDEKI ANDO † and TOSHIO SHIMADA †

It is very important for current processors to effectively exploit an instruction cache. This paper proposes a code placement scheme to reduce instruction cache misses. Our experiments in m88ksim and perl from SPECint95 benchmark suits show that our scheme reduces cache miss rate from 4.14% to 0.00% and from 3.81% to 1.79% with an 8Kbyte instruction cache, respectively. These reductions are large enough to significantly improve performance. Performance of a four-issue superscalar processor is improved by 61% in m88ksim and 23% in perl.

1. はじめに

プロセッサの処理速度は急激に向上しているが，メインメモリの速度向上は小さく，両者の速度差は年々大きくなっている。これに伴ってキャッシュミスペナルティがプロセッサの性能に与える影響も大きくなっている。特に，近年のプロセッサはパイプライン段数，命令発行幅ともに大きくなっており，高い命令供給量が要求されるため，命令キャッシュミスによる命令供給の停止は深刻な問題となる。このため，命令キャッシュを有効利用して，キャッシュミスを削減させることが非常に重要となっている。

キャッシュミスを原因により分類すると，はじめての参照により発生する**初期参照ミス**，キャッシュ容量の不足により発生する**容量性ミス**，1 つのキャッシュラインにマッピングされた複数のメモリラインが互いに入れ替わることにより発生する**競合性ミス**の 3 つが挙げられる[4]。このうち，容量性ミスと競合性ミスはコンパイラによるコード配置により回避することができると考えらえる。本論文では，競合性ミスを減少させるコード再配置手法を提案する。

これまでの研究では，コード再配置の多くは関数単位で行なわれてきた[1),3),7)]。しかし，関数の内部を細かく見ると，競合に関わる基本ブロックや競合に関わらない基本ブロックが存在するので，競合に関わる基本ブロックの配置が競合に関わらない基本ブロックによって阻害されるという問題がある。このような問題を回避するには，基本ブロック単位での配置を行なう必要がある。

基本ブロック単位によるコード再配置の研究もこれまでに行なわれている[5),9)]。しかし，これまでの手法は競合の回避を十分に考慮して配置を行なっている訳ではない。

本アルゴリズムは，これまでの研究において最もキャッシュミス率を削減している Gloy らによる**時間順序情報** (Temporal Ordering Information) を用いた手法[1)]を基本として用いる。ただし，単純にこの手法を基本ブロックに適用すると，1 つの基本ブロックが 1 つのキャッシュラインを占有するため，キャッシュ内に未使用領域が多数発生し，キャッシュの利用効率が低下する。このために，本アルゴリズムでは制御フローグラフ上で近傍に位置する複数の基本ブロックをまとめて，キャッシュラインに近い大きさのブロックにし，このブロックを単位として Gloy らのアルゴリズムを適用する。

本アルゴリズムにより，8K バイトの命令キャッシュにおいて，SPECint95 の m88ksim において 4.14% であったキャッシュミス率を 0.00% に，perl において 3.81% であったキャッシュミス率を 1.79% に，大幅に削減することに成功した。これは，4 命令発行のスーパースカラプロセッサの性能をそれぞれ 1.61 倍，1.23 倍も向上させることができる。

2 章では，過去に行なわれたコード再配置の研究について述べる。3 章では，既存の関数再配置手法を基本ブロックに適用した場合の問題点を挙げる。4 章では，我々のアルゴリズムの詳細を説明する。5 章で本手法の評価と検討を行ない，6 章で本研究のまとめを行なう。

2. 関連研究

現在まで，キャッシュヒット率向上のためのコード再配置に関する研究が行なわれてきた。

Pettis と Hansen らは，関数間の呼び出し関係を解析し，関数を再配置する手法を提案した[7)]。この手法では，関数をノード，関数同士の呼び出し関係を辺とし，呼び出しが行なわれた回数をその辺の重みとした呼び出しグラフを作成する。最も重みの大きい辺の両端の関数をキャッシュ上で隣接するように配置することにより，関数間の競合を減少させている。

Hashemi らは，**Cache Line Coloring** と呼ばれる関数の再配置手法を提案した[3)]。この手法も Pettis と Hansen らの手法と同様，重み付き呼び出しグラフを用いており，最も重みの大きい辺の両端の関数が隣接するように配置を行なう。Cache Line Coloring は Pettis と Hansen の手法とは異なり，キャッシュ容量と関数のサイズを考慮することで，同一キャッシュライン上に配置された関数同士の競合を検出することが可能である。さらに競合を回避するために一度配置が決定した関数の配置の変更を行なっている。ベンチマークに SPECint95, SPECint92 を用い，8K バイトのキャッシュでこの手法を評価した結果，キャッシュミス数が元の配置と比較して 40%, Pettis と Hansen の結果と比較して 17% 削減された。

Gloy らは，時間順序情報と呼ばれるプロファイル情報を用いて，関数の再配置を行なう手法を提案した[1)]。この時間順序情報とキャッシュや関数のサイズを用いて配置を行なうことにより，Pettis と Hansen に比べて 0.3% ～ 1.1%, Hashemi の結果に比べて 0.4% ～ 1.1% キャッシュミス率が減少した。

これらの手法はすべて関数の再配置であり，関数内の基本ブロックは元の配置と同じである。そのため，キャッシュヒット率

† 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University

を最大限に引き出しているとはいえない。

Hwu らは, IMPACT-I コンパイラで以下のような手法を採用している[5]。

- 関数内の連続して実行される複数の基本ブロックを一つにまとめる (これをトレースと呼ぶ)。
- 各関数内で, 連続して実行されるトレースを隣接するキャッシュラインにマッピングされるように配置する。
- 連続して実行される関数が同じキャッシュラインにマッピングされないように, 関数を配置する。

この手法では, トレースを作成することで空間的局所性を活用することができるが, 競合を回避するためのトレースの配置の範囲が各関数内に制限されているため, キャッシュミス率を最小限にすることはできない。

冨山らは, プログラムコードをトレースに分割し, トレースの配置問題を定式化したものを線型計画法によって解くことにより, キャッシュミスが最小となるコード配置を求める手法を提案した[9]。この手法により, 1K バイトのキャッシュにおいて, GNU grep でキャッシュミス数が 45.5% 減少した。この手法では, 関数の境界を越えて基本ブロックを配置しているが, 線型計画法により配置を決定するために計算量が非常に多く, ごく小規模なプログラムにしか適用することができない。

本論文では, 基本ブロック単位で配置を行なうことが可能で, 実用的な規模のプログラムにも適用することが可能なアルゴリズムを提案する。

3. 既存の手法による配置

既存の手法による配置として, Cache Line Coloring と Gloy のアルゴリズムの考え方を基本ブロック単位でのコード再配置に用いた場合を, 例を挙げて説明する。図 1 のプログラムを, 4 命令の容量で, ラインサイズ 2 命令のキャッシュへ配置することを考える。

```
for (i = 0 ... 99) {
    block A;
    if (i < 60) {
        block B;
    } else {
        block C;
    }
    block D;
}
```

ブロック	命令数
A	1
B	2
C	2
D	1

図 1　プログラムの例

このプログラムに対する制御フローグラフは図 2 のようになる。

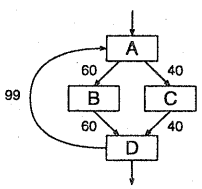

図 2　プログラム例に対する制御フローグラフ

3.1 Cache Line Coloring

Cache Line Coloring の考え方を基本ブロック単位の再配置に適用する場合, 呼び出しグラフの代わりに制御フローグラフを用いればよい。制御フローグラフの辺の重みが大きいブロックがキャッシュライン上で隣接するように配置し, 競合を回避する。Cache Line Coloring による配置の流れを図 3 に示す。図中の破線は, キャッシュへのマッピングが分かりやすいように記してある。

最初に, 最も重みの大きい辺とその両端のノードである基本ブロック A と D を選択し, 図 3 (a) のようにキャッシュ上で隣接するように配置する。次に重みの大きい辺は A と B を結ぶ辺であるが, A は既に配置されており, B は A, D のどちらとも競合するので, 競合しない配置はない。そこで, 例えば図 3 (b) のように配置するとする。同様に C の配置を行なうが, C も A と D との競合を避けることができないので, 空いているラインに配置する。最終的には, 図 3 (c) のような配置となる。

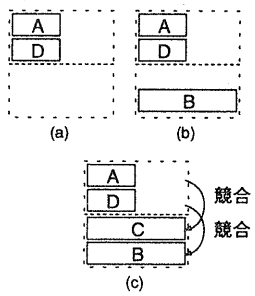

図 3　Cache Line Coloring による配置

この手法では 2 つのラインで競合が発生している。

3.2 Gloy らのアルゴリズム

Gloy らのアルゴリズムではキャッシュの時間的な挙動に注目した時間順序情報と呼ばれる情報を用いて再配置を行なう。Gloy らのアルゴリズムにおいて, 関数を基本ブロックに置き換えて時間順序情報を説明すると以下のようになる。

キャッシュがヒットするためには, あるブロック i が参照されてから再び参照された時, ブロック i が他のブロックによってキャッシュから追い出されていない必要がある。例えば, ブロック i の 2 回の参照の間にブロック j の参照が必ず起こるとすると, i と j が同一キャッシュラインにマッピングされた場合に, 常に競合を起こす。「ブロック i の連続する 2 回の参照の間にブロック j の参照が起こる」という事象が発生する頻度が, ブロック i, j 間の競合の度合として, 競合の回避に用いられる。このような, 2 つのブロック間の競合の度合を示すものが時間順序情報である。

図 1 のプログラムでは, ブロック B が先に 60 回実行された後, ブロック C が 40 回実行されるので, B と C の間では競合は起こらない。時間順序情報を用いると, B と C が競合を起こさないことが検出できるので, 図 4 (a) のように B と C を同一のキャッシュラインにマッピングされるように配置しても良いことが分かる。そうすれば, 図 4 (b) のような配置になる。

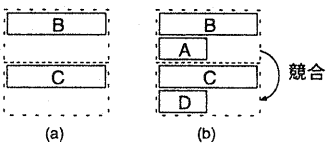

図 4　時間順序情報を用いた配置

この手法を用いると，競合が発生するラインが 1 つに減少する☆。

3.3 既存の手法の問題点

上記 2 つの手法のいずれも，競合を起こすキャッシュラインが存在する。しかし，理想的には図 5 のような配置を行なえば，競合が起こらない。

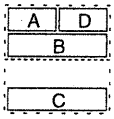

図 5　競合を起こさない配置

既存の方法を単純に基本ブロックに適用すると，1 つの基本ブロックが 1 つのキャッシュラインを占有するため，キャッシュ内に未使用領域が発生し，キャッシュの利用効率が低下する。さらに，これは競合を回避するための再配置を阻害する。しかし，図 5 のように，基本ブロック A と D を同一ラインにまとめることにより競合を回避することができる。

次章では，このような問題点を解消したアルゴリズムを説明する。

4. アルゴリズム

4.1 方　　針

関数再配置によるアルゴリズムを単純に基本ブロックの再配置に用いた場合，キャッシュの未使用領域の発生が問題となっていた。本アルゴリズムでは未使用領域の発生を抑えるため，複数の基本ブロックをまとめて，キャッシュラインサイズに近い大きさのブロックを作成する。これをマージブロックと呼ぶ。この際，できるだけ近い時間に参照されるブロックの組を選択し，マージブロックを作成すると，空間的局所性を有効に利用することができる。このマージブロックを最小単位として，Gloy のアルゴリズムを適用する。

4.2 節では，時間順序情報とブロック間の距離の情報の採取方法について述べる。4.3 節では，マージブロックの作成方法について述べる。4.4 節では，マージブロックを配置する方法について述べる。

4.2 プロファイルの採取

プロファイル情報として，次の 2 つを採取する。

- 基本ブロック i が参照されて，再び i が参照されるまでに，基本ブロック j を参照するという事象が発生した回数 $n_{i,j}$
- 基本ブロック j を参照してから，基本ブロック i に到達するまでに参照された基本ブロックの数の合計 $d_{i,j}$

前者は時間順序情報で，$n_{i,j}$ の値が大きいほど 2 つのブロック i, j が同一のキャッシュライン上に配置されると，多数の競合を起こす。この情報は，ブロックの配置の決定に用いる。

後者は基本ブロック同士の時間的な距離の情報である。ブロック i から j までの時間的距離は $\frac{d_{j,i}}{n_{i,j}}$ で求めることができる。この情報は我々のアルゴリズムで新たに導入したプロファイルで，マージを行なうブロックの組の決定に用いられる。

以下では，具体的な例を用いてプロファイルの採取方法を説明する。

プロファイル採取には，図 6 に示すように基本ブロックの実行履歴を保持するためのスタックと，$n_{i,j}$, $d_{i,j}$ を記録するための配列を用意する。

プログラム中の基本ブロックが図 7のような順序で実行されるとする (数字は基本ブロックに割り振った番号)。

スタックには，基本ブロック番号が実行された順に積まれて

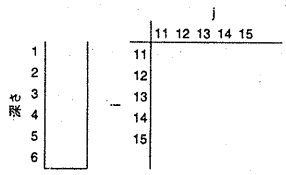

図 6　スタックと配列

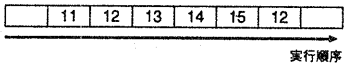

図 7　基本ブロックの実行トレース

いく。基本ブロック 11 から 15 までが実行されると，図 8 の状態になる。

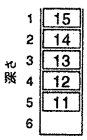

図 8　スタックの状態 (ブロック番号 15 実行直後)

次にスタックに積まれるブロック番号は 12 であるが，ブロック番号 12 は既にスタック内に存在する。この場合，図 9 に示すように，スタックの先頭からブロック番号 12 までのブロックに対応する $n_{i,j}$ には 1 を加算する。つまり，$i = 12$, $j = 12, 13, 14, 15$ に対する $n_{i,j}$ に 1 を加算する。同様に $d_{i,j}$ に対しては，各ブロックに対応するスタックの深さの値を加算する。この例では，$d_{12,13}, d_{12,14}, d_{12,15}$ にはそれぞれ 3, 2, 1 を加算する。

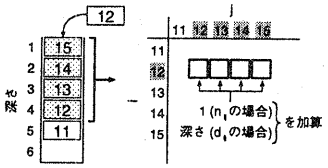

図 9　$n_{i,j}$, $d_{i,j}$ への加算

そして，既にスタック内に存在していたブロック番号 12 を削除し，スタックの最上部にブロック番号 12 を積む (図 10)。

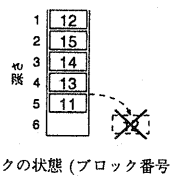

図 10　スタックの状態 (ブロック番号 12 再実行後)

以上の手順をプログラムの終了まで繰り返し，$n_{i,j}$, $d_{i,j}$ を採取する。

4.3 マージブロックの作成

マージブロックの作成は，以下のような手順で行なう。

☆ ただし，この例ではキャッシュミス数は Cache Line Coloring の場合と変わらない。

(1) 全ての基本ブロックの末尾に無条件分岐命令を追加する☆。

(2) 実行回数の多いブロック A を選択し，これをマージブロック A とする。

(3) キャッシュラインのサイズから現在のマージブロック A のサイズを引いた値よりも小さいサイズの基本ブロックで，$\frac{d_{B,A}}{n_{A,B}}$ が最小となる基本ブロック B を，現在のマージブロック A の末尾に付加する。これを可能な限り繰り返す。

(4) 全ての基本ブロックがマージブロックに入るまで (2) ～ (3) を繰り返す。

ブロック A にブロック B をマージする手順 (3) においては，以下のようにプロファイル情報もマージする (図 11 参照)。

$$n_{A,x} = n_{A,x} + n_{B,x}$$
$$n_{x,A} = n_{x,A} + n_{x,B}$$
$$d_{A,x} = d_{A,x} + d_{B,x}$$
$$d_{x,A} = d_{x,A} + d_{x,B}$$

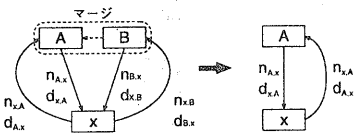

図 11 プロファイル情報のマージ

4.4 マージブロックの配置

マージブロックの配置は以下のように行なう。

(1) 配置されていないブロックの中から最も実行回数の多いものを選択する。

(2) キャッシュラインを走査し (図 12)，各キャッシュライン毎に既に配置されているブロックと起こすキャッシュミスを計算する。

(3) 計算したミスが最も小さいラインへブロックを配置する。

(4) 全て配置されるまで (1) ～ (3) を繰り返す。

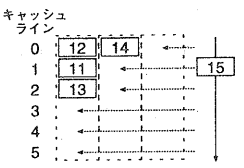

図 12 キャッシュラインの走査

手順 (2) におけるキャッシュミスの計算方法はダイレクトマップ方式の場合，以下のように求める。

まず，ブロック i が参照されてから再び参照されるときに，ブロック j によって i がキャッシュから追い出されていない確率は $1 - \frac{n_{i,j}}{n_{i,i}}$ である。i と同一キャッシュラインにマッピングされるブロックの集合を B とすると，B に含まれるブロックによって i が追い出されていない確率は次の式で表わされる。

$$\prod_{j \in B \neq i} (1 - \frac{n_{i,j}}{n_{i,i}})$$

i の参照によって起こるキャッシュミス数は，(i の実行回数) × (i が追い出されている確率)，つまり次の式で表わされる。

$$n_{i,i}\{1 - \prod_{j \in B \neq i} (1 - \frac{n_{i,j}}{n_{i,i}})\}$$

これらより，集合 B 全体によって起こるキャッシュミス数は次の式で表わされる。

$$\sum_{i \in B} n_{i,i}\{1 - \prod_{j \in B \neq i} (1 - \frac{n_{i,j}}{n_{i,i}})\}$$

この式を用いて，キャッシュミス数の見積もりを行なう☆☆。

5. 評　　価

本章では，提案した手法によるキャッシュミス率の低減を評価する。さらに，その低減により，プロセッサの性能をどれほど向上させることができるのかを評価した。

5.1 キャッシュミス率

5.1.1 評価方法

キャッシュミス率を測定する対象アーキテクチャとして DEC Alpha 21164[2] を用い，プログラム解析ツール ATOM[8] を用いたキャッシュシミュレーションにより測定を行なった。ベンチマークには SPECint95 の中から m88ksim と perl の 2 つを用いた。compress95, ijpeg, li は，コードのワーキングセットが小さく，再配置を行なわなくても十分命令キャッシュミスが小さいため，対象ベンチマークから除外した。ベンチマークのコンパイルは GNU GCC 2.7.2 で行なった。

対象となるキャッシュは DEC Alpha 21164 に搭載されているものと同じ 32 バイトラインのダイレクトマップ方式で，キャッシュ容量を 1K バイトから 8K バイトまで変化させながら，再配置を行なわないプログラムと再配置後のプログラムを実行した場合のキャッシュミス率を測定した。表 1 に使用ベンチマーク，プロファイル採取とキャッシュミス率の測定に用いた入力ファイル，静的命令数，動的命令数を示す。

5.1.2 結　　果

本アルゴリズムによって再配置を行なったコードを実行した場合の命令キャッシュミス率と，再配置を行なっていないコードを実行した場合の命令キャッシュミス率を表 2 と図 13 に示す。

表 2 から分かるように，キャッシュ容量が 2K ～ 8K バイト の場合には，m88ksim の場合には命令キャッシュ

☆ 無条件分岐で終了している基本ブロックでは，さらに無条件分岐命令を追加する必要はないが，現在の実装ではこのような判断は行なっていない。

☆☆ Gloy のアルゴリズムでは，$\sum_{j \in B \neq i} n_{i,j}$ としているが，我々の計算方法の方がより正確である。

表 1　ベンチマークの詳細 (キャッシュミス測定)

ベンチマーク	プロファイル			測定		
	入力	静的命令数	動的命令数	入力	静的命令数	動的命令数
m88ksim	dcrand.lit (train)	56K	76M	dhry.lit (ref)	68K	390M
perl	scrabbl.pl (train)	119K	52M	primes.pl (ref)	142K	19G

表 2　再配置前と再配置後の命令キャッシュミス率

キャッシュ容量 [バイト]	命令キャッシュミス率 [%]			
	m88ksim		perl	
	再配置前	再配置後	再配置前	再配置後
1K	8.72	10.55	14.02	13.58
2K	5.42	1.46	10.18	8.75
4K	4.15	0.71	7.44	5.23
8K	4.14	0.00	3.81	1.79

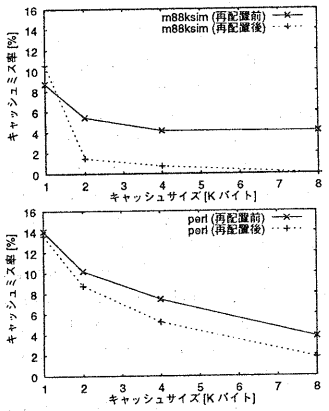

図 13　再配置によるキャッシュミス率の変化

ミス率を大幅に減少させている。perl の場合においても, 1% 以上キャッシュミス率を減少させている。DEC Alpha 21164 に用いられているものと同様の 8K バイトの容量の場合, m88ksim において再配置後のミス率が 1 % 以下と, 極めて小さい値にまで減少させることができた。

文献 1) によれば, Gloy らの関数再配置による手法では, 8K バイトのキャッシュに対するキャッシュミス率は m88ksim で 2.88%, perl で 2.50% である。ただし Gloy らと我々とでは以下の測定条件が異なっている。

- Gloy らは perl のコンパイルに SUIF コンパイラ (バージョン 1.1.2) を用いている (m88ksim は我々と同じ GNU GCC 2.7.2 を用いている)。
- プロファイルの採取とキャッシュミス率の測定において, Gloy らは m88ksim の実行ブロック数を 50M に制限し, perl の入力として規模の小さいものを用いているが, 我々は制限を行なっていない。

以上のような条件の違いがあるため, 公平に比較はできないが, 基本ブロック単位による配置の優位性は十分に認められる。

一方, 命令キャッシュ容量が非常に小さい 1K バイトの場合, m88ksim の再配置後のコードの方が命令キャッシュミス率が大きくなる。perl についても, 再配置前と再配置後のキャッシュミス率の差が小さくなる。これは, 各基本ブロックの末尾に無条件分岐命令を追加したことによるコード量の増加が原因であると考えられる。

表 3　コードの増加量

ベンチマーク	コード量		コードの増加率
	再配置前	再配置後	
m88ksim	226512 バイト	271372 バイト	1.20
perl	475256 バイト	569616 バイト	1.20

表 3 にコードの増加量を示す。同図から分かるように, 無条件分岐命令の追加によりコード量が約 20 % 大きくなる。このため, 再配置を行なわない場合に比べてコードが命令キャッシュから追い出される確率が高くなり, ミス率が大きくなったと考えられる。しかし我々は, この影響はキャッシュ容量が極めて小さい場合のみ生じるものであるため, 大きな問題ではないと考える。

5.2　プロセッサ性能

5.2.1　評価方法

DEC Alpha の命令セットで動作するプロセッサのシミュレータを我々は持っていないので, Alpha でのプロセッサ性能の測定は行なえなかった。代わりとして, MIPS R2000[6] の命令セットで動作するスーパースカラシミュレータを用い, 5.1 節で得られたキャッシュミス率を乱数によって人為的に発生させることでプロセッサ性能を測定した。

この測定は, 次の 2 点において実際の性能とは異なる。

- 現実には, キャッシュミスはランダムには発生しない。
- 再配置後のコードには, 元のコードに比べて無条件分岐命令が増加しているが, 本測定ではその影響はキャッシュミス以外には現われない。

以上のような問題はあるが, 命令キャッシュミスが性能に与える影響はおよそ分かる。

シミュレータはトレース駆動方式のものを用いた。シミュレーションに用いたスーパースカラマシンの構成は表 4 の通りである。

キャッシュミスの測定と同様, ベンチマークには m88ksim と perl を用いた。両ベンチマークのトレースは NEC EWS4800/360AD (MIPS R4400) 上で, MIPS R2000 のコードを生成して採取した。トレース長はそれぞれ 120M と 85M 命令である。

表4　マシン構成

命令発行幅	4 命令
機能ユニット	ALU, シフタ, 分岐, ロード, ストア 各 ×4
命令キャッシュ	8K バイト 32 バイトライン キャッシュミスペナルティ 6 サイクル キャッシュミス率は指定可能
データキャッシュ	完全
L2 キャッシュ	完全
分岐予測	gshare 予測機構 64K エントリ 履歴長 16 予測ミスペナルティ 5 サイクル
BTB	完全

5.2.2 結　　果

図 14 に測定結果を示す。再配置前と再配置後に加えて, キャッシュミスが起こらないと仮定した場合についても測定を行なった。

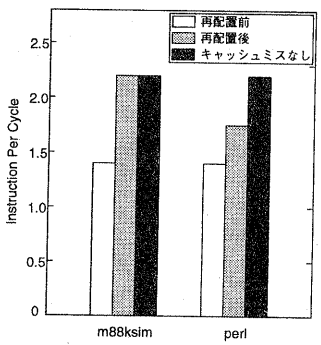

図 14　命令キャッシュヒット率に対する IPC の変化

再配置前に対して, 再配置後の性能向上は m88ksim では 1.61 倍, perl では 1.23 倍となっている。この結果より, キャッシュミスを減少させることにより, 極めて大きくプロセッサ性能を改善できることが分かる。

6. ま と め

本稿では, 実用的なアプリケーションに適用可能な, 基本ブロックレベルのコード再配置手法を提案した。この手法を用いることにより, 8K バイト の命令キャッシュの場合, m88ksim の命令キャッシュミス率が 4.14% から 0.00% に, perl の命令キャッシュミス率が 3.81% から 1.79% に減少した。この減少率は, プロセッサの性能を著しく改善するだけの量であり, 4 命令発行のスーパースカラプロセッサの性能を m88ksim で 1.61 倍, perl で 1.23 倍に向上させることができる。今後, 他のベンチマークについても測定を行ない, 有効性を確認していく予定である。

謝　　辞

本研究の一部は, 文部省科学研究費補助金基盤研究 (C)「広域命令レベル並列によるマイクロプロセッサの高性能化に関する研究」(課題番号 1068034) の支援により行なった。

参 考 文 献

1) N.Gloy, T.Blackwell, M.D.Smith, and B.Calder, "Procedure Placement using Temporal Ordering Information," In *Proceedings of 30th Annual IEEE/ACM International Symposium on Microarchitecuture*, pp. 303–313, December 1997.
2) L. Gwennap, "Digital Leads the Peak with 21164," *Microprocessor Report*, Vol. 8, No. 12, September 1994.
3) A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient Procedure Mapping using Cache Line Coloring," In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 171–182 June 1997.
4) M. D. Hill, "*Aspects of Cache Memory and Instruction Buffer Performance*," Ph.D. thesis, University of California at Berkeley Computer Science Division, 1987.
5) W.W. Hwu and P.P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," In *Proceedings of 16th International Symposium on Computer Architecture*, pp. 242–251, May 1989.
6) G. Kane, *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
7) K. Pettis and R. Hansen, "Profile Guided Code Positioning," In *Proceedings of ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16–27, June 1990.
8) A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysys Tools," In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196–205, June 1994.
9) H. Tomiyama and H. Yasuura, "Size-constrained Code Placement for Cache Miss Rate Reduction," In *Proceedings of 9th International Symposium on System Synthesis*, pp. 96–101, November 1996.