システムソフトウェアと
オペレーティング・システム　89－2
シ　ス　テ　ム　評　価　2－2
（２００２．２．１４）

SPEC CINT2000(181.mcf) の縮小プログラム開発手法とその評価

小野寺　聡[†]　上田晴康[†]　安里　彰[†]

計算機アーキテクチャの研究・開発において、シミュレータによる評価は不可欠であるが、実行速度の問題から、SPEC2000 等の標準ベンチマークを使用することは難しい。我々は、CINT2000 の 181.mcf をターゲットとし、オリジナルプログラムのメモリアクセスパターンや命令コンビネーションといった特性を保持しつつ、実行ステップ数を大幅に減少させて約 1/700 にした縮小プログラムを開発した。181.mcf は、縮小対象のループにおいて動的にデータが変動し、それに伴ってプログラムの挙動が変化する。そのため、CFP2000 のプログラムとは違った作成手法を用いる必要がある。本稿では、その作成手法と評価について述べた。

Development and Evaluation of the Shrunk Program of SPEC CINT2000(181.mcf)

Satoshi Onodera,[†] Haruyasu Ueda[†] and Akira Asato [†]

It is neccessary to use some simulators to evaluate computer architectures. But standard benchmarks such as SPEC2000 are not usually used on simulators in terms of execution time. We developed the shrunk program of CINT2000(181.mcf). Although the shrunk program has about 1/700 of execute steps of original program, it has same characteristics such as memory access pattern and combination of instructions. Because behaviors of the core loop in 181.mcf change dynamically, we have to consider a different development policy compared with some programs of CFP2000.

1. はじめに

計算機アーキテクチャの研究・開発において、論理シミュレータやアーキテクチャシミュレータを使用した性能評価は不可欠である。しかしながら、シミュレータの実行時間は実機の数百倍から数万倍にもおよぶため、実機の性能評価に通常使用されるような実行時間の長いベンチマークを用いて評価を行うのは難しい。

この問題を解消する手段として、サンプリング抽出や類似命令の抽出といった研究が行われている[1)2)3)4)]。ここで、我々はオリジナルプログラムのソースコードを変更し、メモリアクセスパターンや命令コンビネーションといった特性を保持したままで、実行時間を短くした縮小プログラムを開発する手法を提案する。この縮小プログラムを用いれば、シミュレータ上で前述のようなベンチマークの評価を行うことができ、計算機アーキテクチャの研究・開発に大きく貢献することができる。

縮小プログラム対象ベンチマークとしては、標準ベンチマークとして使用されている SPEC2000[5)] を考える。SPEC2000 は、CFP2000 と CINT2000 に分けられる。CFP2000 のプログラムは、プログラム構造が規則正しく、データ構造も動的に変化しないものが多いので縮小方針がたてやすい[6)]。しかしながら、CINT2000 のプログラムでは動的にデータが変動し、それに伴ってプログラムの挙動が変わったり[7)]、コアループのキャッシュミス数がキャッシュ構成によって変動することがある。そのため、CFP2000 のプログラムと同様の手法では縮小プログラムを作成できない。

本論文では、CINT2000 のプログラムの中から 181.mcf[8)] を選び、縮小プログラムを作成した。作成した縮小プログラムの評価は、命令コンビネーションとキャッシュミス率から行った。

本論文の構成は以下の通りである。まず 2 章では mcf プログラムの概要について説明する。次に 3 章では縮小プログラムの作成手法について述べる。4 章では作成した縮小プログラムの評価を行い、最後に 5 章でまとめを行う。

2. mcf プログラムについて

2.1 概　　要

mcf は、大規模交通流モデルの最適化プログラムのアルゴリズムを使用したベンチマークである。何台かの車とそれが格納されている車庫、さらに車の出発点と到着点を考える。車は車庫を出て出発点に到着し人を乗せ

† (株) 富士通研究所
FUJITSU LABORATORIES LTD.

る。その後到着点へ行き人を下ろす。次に車は車庫へ入るか、別の出発点に行き人を乗せる。この車の流れをスケジューリングする中で、(1) 使用する車の数を最小にする、(2) 車の移動に必要なコストを最小にする、という 2 つを目的とする。

2.2 データ構造と処理フロー

2.2.1 使用するデータ

mcf では、2n+1 個のノードを仮定する。ノードのうち n 個は出発点で n 個は到着点、1 個は車庫である（図 1(a)）。車はこれらのノード間を移動し、この移動を trip と呼ぶことにする。

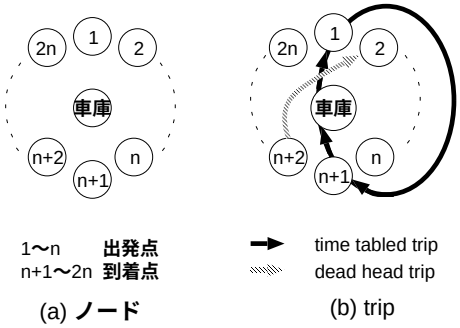

図 1 mcf の (a) ノード、(b)trip の例

基本の trip は、(1) 車庫 → 出発点、(2) 出発点 → 到着点、(3) 到着点 → 車庫の移動から成る。これらの trip は time tabled trip と定義される。一方、この time tabled trip を結ぶ役割を果たす trip も存在し、dead head trip と定義されている。これは到着点から出発点を結ぶ trip のことである。これら 2 種類の trip を図 1(b) に示す。time tabled trip は 3n 通り存在するのに対し、dead head trip は n^2 通り存在する。入力データとしては、全ての time tabled trip と、dead head trip の一部がある（dead head trip はプログラム内で生成し、追加される）。また、node と trip それぞれに対するデータは、node データと arc データとして構造体配列が定義される。node データには通過する時間や次に向かうノードへの trip のデータが、arc データには trip の出発点と到着点、必要なコストの情報が保持されている。

2.2.2 処理フロー

処理フローは以下のようになっている。

(1) 初期値の代入
(2) ノードの構造をスケジューリングし、変更する
(3) 新しい dead head trip の生成
(4) (3) で新しく trip が生成されたら (2) へ、そうでなければ終了

まず、初期状態として、node データは図 2(a) のようなデータ構造を持っている（ノード 0 は車庫を意味する）。さらに、スケジューリングが行われると、図 2(b) のような構造へと変化する。スケジューリングが終了すると、新しく arc データが生成される。この新しく作成されたデータを追加して、2 回目のスケジューリングが行われる。この際、1 回目と較べてデータ総量が増えているためにプログラムの挙動が変わる。

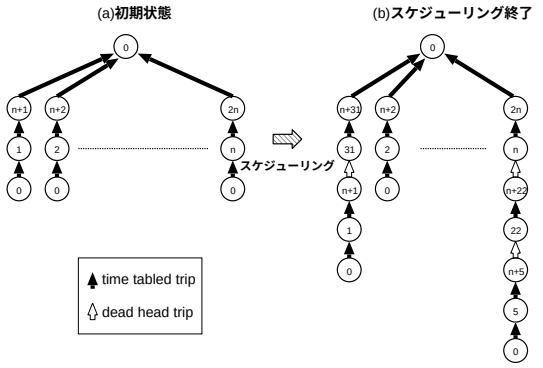

図 2 スケジューリングによるデータ構造の変化

何度かスケジューリングを行った後、車の数および移動コストが最小になったら処理を終了する。この判断は、新しく生成される dead head trip の数が 0 であることから行う。

2.3 mcfプログラムの構成

mcf のソースコードの概要を、図 3に示す。

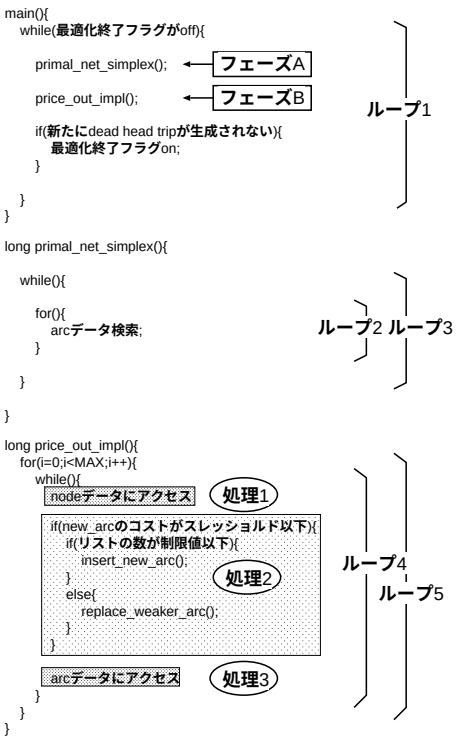

図 3 mcf のソースコード概要

main() 関数内の while ループが最外ループである。その中で関数 primal_net_simplex() と関数 price_out_impl() が実行されている。以下簡単のため、図 3のようにコアとなるループをループ 1 ～ 5 とし、 primal_net_simplex() および price_out_impl() が実行されているフェーズを、それぞれフェーズ A とフェーズ B と呼ぶことにする。

オリジナルの mcf では、ループ 1 を 10 回繰り返す。ループ 1 はフェーズ A とフェーズ B から成り、フェーズ A ではその時点で存在する trip の中から各ノードに対して最適な trip をスケジューリングし、フェーズ B では、新しく dead head trip の arc データを生成する。

フェーズ A と B については、個別に挙動を説明する。

2.4 フェーズ A の挙動

フェーズ A はループ 2 と 3 で 2 重ループになっている。ループ 2 の挙動を簡単に図 4に示した。 46.2MB のデータ領域を、 150KB ずつのブロック（0 ～ 299）にわけ、まずブロック 0 の 1 番目のデータにアクセスする。次にブロック 1 の 1 番目のデータにアクセスする。この作業を最後のブロックまで行った後、 1 番目のブロックに再び戻り、 2 番目のデータにアクセスする。この処理を順次繰り返す。

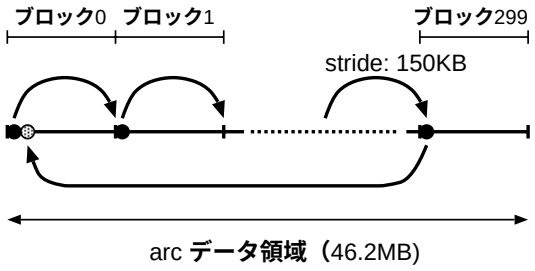

図 4 フェーズ A の挙動

2.5 フェーズ B の挙動

フェーズ B では、ループ 4 と 5 の 2 重ループとなっている。外側のループ 5 のインデックスの値が増加すると、内側のループ 4 の回転数も増加する。ループ 4 では、新しく dead head trip の arc データを生成する作業を行っているが、ループ 1 の 1 回のイタレーションで生成できる arc の数は、 MAX_NEW_ARCS という定数で決定されている。この数を越えて生成された場合は、よりコストの低い arc データを採用する。ループ 4 は処理 1 ～ 3 から成っているが、処理 1 と 3 では新しく arc データを生成するために node データおよび arc データへとアクセスを行う。

処理 2 では、よりコストの小さい dead head trip を生成するため、以下のような作業を行っている。

(1) 新しく dead head trip の arc データ（new_arc）を生成し、そのコストがあるスレッショルド以下かどうかをチェックする。

(2) (1) の条件を満たし、かつリストに追加された new_arc の数が MAX_NEW_ARCS 以下ならば、リストに追加する。

(3) (1) の条件を見たし、かつリストに追加された new_arc の数が MAX_NEW_ARCS 以上だったならば、リストの arc データとコスト比較を行い、置き換えを行う。

上記の作業において、 (2) では insert_new_arc() という関数を呼び出す。これは、 new_arc をリストに追加する関数である。リストに入る arc データの数は MAX_NEW_ARCS で決まっており、この値を越えるまでは (1) の条件を満たした new_arc はリストに追加される。リストに追加されると、ソートを行ってコストの大きいものから順に並べられる（図 5(a)）。

(2) で追加した arc データの数が MAX_NEW_ARCS に至ると、 (1) の条件を満たす new_arc が見つかっても (2) は実行されない。代わりに (3) が実行される。 (3) では、 replace_weaker_arc() という関数が実行される。これは、 (1) で生成したリストの一番上にある arc データのコストと、新しく生成された new_arc のコストを比較し、新しく生成した方のコストが低ければ、リストの先頭の arc データと置き換えを行う。置き換えを行った後、再びリストをソートする（図 5(b)）。

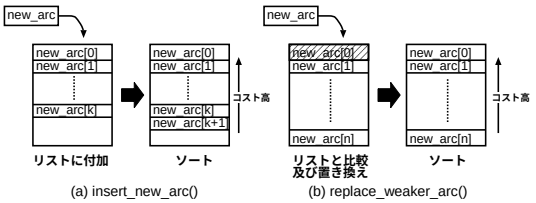

図 5 処理 2 で呼び出す関数

ループ 4 内で処理 2 の (1) の条件を満たす new_arc の数は、ループ 5 のループインデックス i の値が増加しても、それと連動せずに変化する。

また、処理 2 が実行された時の実行ステップ数は、動的に変化するリストの内容によって異なる。 for ループインデックス i が小さい時は、リストの数が少ないために、 (1) を満たした new_arc は全てリストに追加される（insert_new_arc()）。 i がある程度大きくなると、リストに追加できなくなるために、リストの先頭の arc データとの置き換えが行われる（replace_weaker_arc()）。この置き換えによって、リストの先頭の arc データのコストが小さくなり、段々と new_arc との置き換えも起きなくなってくる。そのため、 i がかなり大きくなると、どちらの関数も呼ばれなくなる。これら 3 通りの挙動によって、処理 2 の 1 回の実行ステップ数は異なってくる。

すなわち、ループ 5 のある 1 回のイタレーションで実行されるループ 4 での、 (1) 処理 2 の実行回数と (2) 処

理2の実行ステップは、ループ 5 のインデックス i に連動せず、動的に変化する。

2.6 mcfプログラムの特性について

Fujitsu PRIMEPOWER GP7000F を使用し、 mcf プログラムの特性データを採取した。GP7000F の CPU は SPARC64 GP であり、キャッシュ構成は表 1のようになっている。

1 次命令キャッシュ	64KB 4way(Block 64B)
1 次データキャッシュ	64KB 4way(Block 64B)
2 次キャッシュ	2MB direct mapped(Block 64B)

表 1 SPARC64 GP のキャッシュ構成

mcf のプログラムの特性を明らかにするため、メモリアクセスデータおよび高頻度実行部の調査を行った。実行バイナリの作成にあたっては、ソースコードは CPU2000 のバージョン 1.10 を使用し、 SUN Forte6 でコンパイルした。

メモリアクセスデータは SPARC64 GP 内蔵の PA（Performance Analyser）を用い、キャッシュミスデータを中心に表 2にまとめた。 CPI は約 6、命令あたりの 2 次キャッシュミス率は 6.8% である。

	PA
実行ステップ数	58,051,566,369
CPI	6.071320
2 次データキャッシュミス率	6.8466%/I
2 次命令キャッシュミス率	0.0054%/I
2 次キャッシュミス率	6.8519%/I

表 2 メモリアクセスデータ

高頻度実行部の調査は、プロファイラを使用して使用頻度の高い関数を調査し、表 3にまとめた。

	elapsed time	実行比率
price_out_impl()	829.95 sec	42.99%
primal_net_simplex()	586.57 sec	30.38%
primal_bea_mpp()	463.46 sec	24.00%

表 3 関数使用回数データ

表 3において、 price_out_impl() はフェーズ B で使用される関数であり、 primal_net_simplex() と primal_bea_mpp() はフェーズ A で使用される関数である。つまり、フェーズ A とフェーズ B での全体の実行時間の 97% を占めている。

また、 mcf オリジナルプログラムの 2 次キャッシュミス率のグラフを、図 6に示す。ミス率が 4% 程度を示しているところはフェーズ A、 10% 程度を示しているところはフェーズ B に相当しており、ループ 1 が 10 回繰り返されていることがわかる。

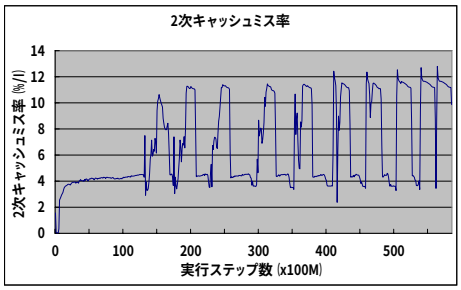

図 6 2 次キャッシュミス率

3. mcf 縮小プログラムの作成手法

3.1 縮小プログラム作成にあたって

オリジナルプログラムの特性を保持しつつ、プログラムの縮小を行う。オリジナルプログラムでは、 CPI が約 6、実行ステップ数は約 58G である。すなわち、実行サイクルは約 348G となる。このプログラムを 2.6 節の GP7000F で実行すると、約 26 分かかる。また、このプログラムを実行時間が 10,000 倍のシミュレータで測定すると、 6ヶ月かかる計算になる。

現実的には、シミュレータの実行時間は数時間以内であることが望ましい。数時間で終了するためには、実行ステップ数は 100M 以下に抑える必要がある。フェーズ A とフェーズ B の実行ステップ数の比率はほぼ 1:1 なので、それぞれのフェーズでの実行ステップ数を 40M 程度にする。

また、 mcf は 2 次キャッシュミス率の変動に特徴があることから、この特性に着目して縮小作業を行う。さらに後述するようにキャッシュ構成によって 2 次キャッシュミス率の特性が変化することを考慮に入れ、ダイレクトマップ構成と 2way 構成のそれぞれに対して縮小プログラムを作成する。ダイレクトマップ構成の縮小プログラムは、実機の SPARC64 GP の PA で評価を行う。このため、ターゲットアーキテクチャのキャッシュサイズは 2MB とする。また 2way 構成の評価はシミュレータで行うが、キャッシュサイズは同様に 2MB とする。

作成した縮小プログラムの評価は、メモリアクセスパターンと命令コンビネーションの観点から行う。ここでメモリアクセスパターンの評価は、 2 次キャッシュミス率を使用する。

この前提を踏まえ、以下の手順で作業を行う。

(1) ループ 1 の 10 回のイタレーションから 1 回を抜き出す。
(2) フェーズ A の縮小作業を行う。
(3) フェーズ B の縮小作業を行う。

3.2 ループ 1 の抽出

オリジナルプログラムでは、ループ 1 のイタレーションは 10 回である。イタレーションごとに、フェーズ A のキャッシュミス率にはほとんど変動が見られないが、フェーズ B に関してはキャッシュミス率が増加する傾向にある。

このデータを、 PA で採取して表 4に示した。

イタレーション	2 次キャッシュミス率
1 回目	7.3961%/I
5 回目	9.9003%/I
10 回目	10.7249%/I

表 4 PA によるフェーズ B の 2 次キャッシュミス率

このデータから、 2 次キャッシュミス率が平均に近い 5 回目のループ 1 を抽出し、このデータをベースに縮小作業を行うこととした。

3.3 フェーズ A の縮小

図 3に示したように、フェーズ A ではループ 2 と 3 の 2 重ループ構造になっている。内ループのループ 2 では、規則的に arc データを全てアクセスしている。プログラム構造が規則的であるため、 CFP2000 のプログラムに使用した縮小方法[6] を使用できる。この方法は、ループにおいて、キャッシュミス率が変わらないようにしながら、ループの回転数およびアクセスするデータサイズを減らすというものである。

従って、ここではまず内ループであるループ 2 の回転数およびデータサイズを減らす。その後に外ループであるループ 3 の回転数を減らすことにする。

まず、ループ 2 の回転数を減らす。この関数では、2.4 節の図 4に示したように、コアループの中で arc 全体を 300 ブロックに分け、約 150KB のストライドでアクセスする。 46.2MB のデータに順次アクセスを行うため、容量性ミスによるキャッシュミスが起きる。

ここでループ 2 の回転数を減らすと、それに伴ってアクセスするデータ量も変動し、フェーズ A の 2 次キャッシュミス率に影響を与える。ダイレクトマップ構成のキャッシュに対しては、表 5のようなデータが得られた。このデータは PA で測定したものである。

縮小率	データサイズ	キャッシュミス率	誤差
original	46.2MB	4.3976%/I	–
1/2	23.1MB	4.5747%/I	4.0267%
1/3	15.4MB	4.5598%/I	3.6877%
1/4	11.6MB	4.6438%/I	5.5975%

表 5 ループ縮小に伴うキャッシュミス率の変化

ループの回転数を 1/2 もしくは 1/3 まで減らすと、オリジナルとのミス率の誤差が 4% 程度になる。 1/4 に減らすと誤差は 5% 以上になる。そのため、ループの回転数を 1/3 にすることとした。これに伴って、アクセスする arc のデータ量を 15.4MB、ストライドを 50KB とした。（図 7）。

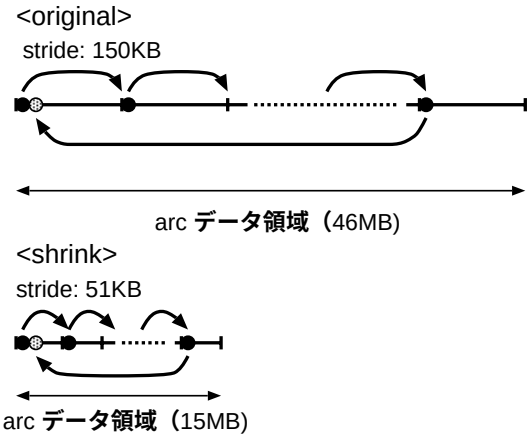

図 7 フェーズ A の縮小

次にループ 3 の回転数を減らす。このループは約 30,000 回回り、その度に縮小したループ 2 を実行する。縮小したループ 2 の 1 回のイタレーションでの実行ステップ数は約 90K であるので、ループ 3 の回転数を 450 回に削減し、フェーズ A での実行ステップ数数を約 40M にする。

なお、フェーズ A のキャッシュミスは容量性ミスに起因するものなので、ダイレクトマップ構成と 2way 構成で、同じ縮小方針を使用するものとする。

3.4 フェーズ B の縮小

3.4.1 フェーズ B のラインコンフリクトについて

フェーズ B のキャッシュミスは、ループ 4 におけるラインコンフリクトによって引き起こされている。このラインコンフリクトは、 2.6MB のデータによって起きているため、キャッシュサイズが 2MB の場合はキャッシュ構成によって挙動が変わることが考えられる。ここで 2MB のサイズを持つ、ダイレクトマップ構成と 2way 構成のキャッシュでの挙動の調査を行った。

ラインコンフリクトを引き起こすデータは node データが 1MB、 arc データが 1.6MB である。これらのデータは、図 8のようにメモリに配置されており、 node データと arc データの先頭は 2MB 離れている。

ループ 4 の先頭と末尾に処理 1 と 3 があり、それぞれ node データと arc データを交互にアクセスする。ループ 5 のインデックス i が増加すると、ループ 4 の 1 回のイタレーションで、処理 1 と 3 によってアクセスするデータ量も増加する。

ここでラインコンフリクトの例として、ループ 5 のインデックス i が 10,000 の時を考える。このとき、600KB の node データと 960KB の arc データに交互にアクセスする。

まず、キャッシュサイズ 2MB、ダイレクトマップ構成のキャッシュを考える。ラインサイズを 64B とすると、キャッシュライン数は 32,768 本になる。 node データの先頭がライン 0 に載ると仮定すると、 node デー

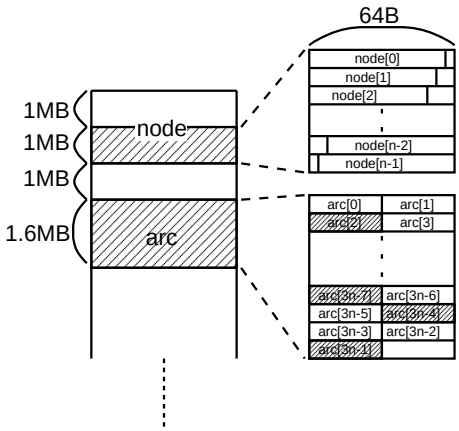

図 8　フェーズ B のデータ配置

タの末尾はライン 9,374 に載る。また、 node データの先頭と arc データの先頭は 2MB 離れているので、このキャッシュ構成の場合では arc データの先頭はライン 0 に載り、末尾はライン 14,999 に載る（図 9(a)）。そのため、ライン 0 〜 9,374 の間では node データと arc データが同じラインを使用する。ここで、ライン 0 〜 9,374 にはまず node データが載り、その後、 arc データがこれらのラインに載る。このため、ラインコンフリクトによるキャッシュミスが起きる。

ループ 5 のインデックス i がインクリメントされると、またループ 4 が実行されるが、この時まず node データがさきほど arc データで使用されたラインに対してキャッシュミスを引き起こす。その後また arc データによってキャッシュミスが引き起こされる。

次に、キャッシュサイズ 2MB、2way 構成のキャッシュを考える。図 9(b) のように 2way 構成になると、アクセスする arc データサイズが 1MB 以下の時はラインコンフリクトが起こらない。1MB までならば、 node データと arc データが同セット内に載るためである。アクセスする arc データのサイズが 1MB を越えると、ラインコンフリクトが起き始める。

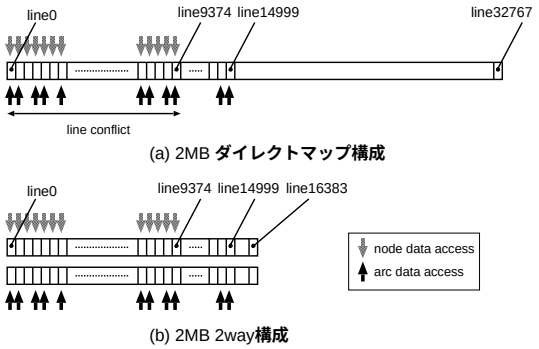

図 9　ラインコンフリクト

ここで、図 10にループ 1 の 5 回目のイタレーションにおける 2 次キャッシュミス率を示した。このデータはソフトウエアシミュレータで測定したものである。

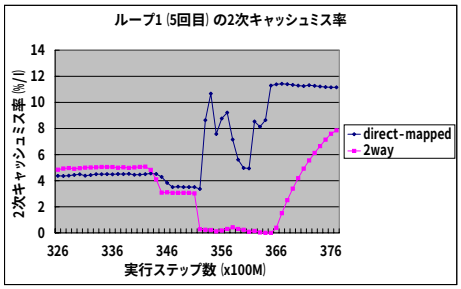

図 10　ダイレクトマップと 2way の 2 次キャッシュミス率

x 軸の 350 が、フェーズ A とフェーズ B の挙動の境目となっている。ここでフェーズ B の挙動は、 2 つのキャッシュ構成で異なっている。すなわち、ダイレクトマップ構成では、フェーズ B が始まってすぐに 10% 近いキャッシュミス率になっているのに対して、 2way ではフェーズ B の最初では殆んどキャッシュミスが起きておらず、 x 軸の 366 を過ぎたあたりから起き始めている。そのため、フェーズ B におけるキャッシュミス率は、ダイレクトマップに比べて 2way 構成では大きく減少することになる。

3.4.2　ダイレクトマップ構成向けの縮小

ここまで示したように、ループ 4 では処理 1 と 3 でラインコンフリクトが起こり、キャッシュミスが多発する。また処理 2 が行われることで、キャッシュミス率の引き下げが行われる。すなわちこのループ 4 におけるキャッシュミス率は、処理 1 と 3 によっておこるキャッシュミス数と、処理 2 の実行命令数によって決まる。

ここで、キャッシュミス数はループ 5 のループインデックス i が増加するにつれて同様に増加する。しかし処理 2 の実行命令数は、 2.5 節に示したように、 i の増加とは連動せずに変化する。そのため、 CFP2000 のプログラムに使用したような方法ではフェーズ B は縮小できない。

例えば、ループ 5 におけるインデックス i の最大値 MAX を小さくすることを考える。この時、ループ 4 の処理 1 と 3 で起きるキャッシュミス数と、処理 2 の実行命令数の合計を測定する。これらの値からキャッシュミス率を計算すると、オリジナルよりもキャッシュミス率が低くなる。これは、ループインデックス i の値が小さい時に、ループ 4 における処理 2 の呼び出し回数および実行命令数が多いためである。

また、ループ 5 を何回かに 1 回ずつ間引いて実行する方法も考えられるが、この方法では実行時のリストの内

容を保持しておかなければ同じ挙動は再現できないため、縮小作業を行うのが難しい。

ここで、ループ 5 の回転数を減らし、さらに処理 2 の実行回数を間引く手法を考える。間引き処理は、ループ 4 内に回転数計測用の変数（cnt）を用意し、この値を用いて処理 2 を数回に 1 回だけ実行する。

まず、フェーズ A の実行命令数が約 40M であることを考慮し、ループ 5 のループインデックスの最大値 MAX を 1/6 に減らした。次に、フェーズ B のソース内に、カウンタ用の変数 cnt を用意し、図 11のようにループ内の処理を書き換えることで、キャッシュミス率をオリジナルに近付けた。

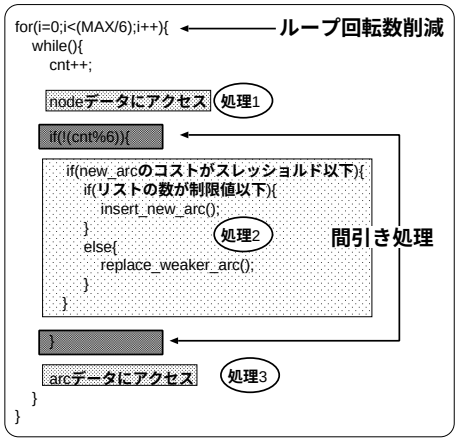

図 11 ループ 4 内の処理 2 の間引き

間引く間隔を変えて調査したところ、 1/6 に間引いた時にオリジナルのキャッシュミス率に最も近付いたため、この値を使用した。

3.5 2way 構成向けの縮小

2way 構成でも基本的な考え方はダイレクトマップ構成と同じである。しかしながら 2way 構成を対象とした時は、ループ 5 の回転数を単純に減らすことはできない。これは 2way 構成の場合には、図 10に示したようにループ 5 の回転数が少ない時にキャッシュミスが起こらないためである。

したがって、フェーズ B においてキャッシュミスが発生し始める場所（図 10の x 軸の 366 より後）からループ 5 を抜き出して、処理 2 の実行回数を間引いた。この場所に対応するインデックスは、だいたい MAX の 4/5 の位置なので、この位置からループ 5 を実行させることにした。また、 2way 構成の場合、オリジナルプログラムにおけるフェーズ B のキャッシュミス率は命令あたり約 3% なので、キャッシュミス回数と処理 2 の実行命令数の比率をこの値に近付けるように縮小を行った。調査した結果、最もオリジナルのキャッシュミス率に近付くのは、間引く間隔を 1/7 にした時であったため、この値を使用することにした。

4. mcf 縮小プログラムの評価

4.1 ダイレクトマップ構成に対する評価

ダイレクトマップ構成向けの縮小プログラムの評価を行った。以下にメモリアクセスと命令コンビネーションのデータを示す。メモリアクセスについては、表 6に PA で採取したデータを示した。 2 次キャッシュミス率の特性に着目して縮小を行ったため、この値に関してはオリジナルと比較すると約 1.8% 程度の誤差で、良好な結果が得られている。また、表 7には命令コンビネーションのデータを示した。こちらもオリジナルと縮小版の比率がほぼ同じであり、縮小がうまくいっていることを示している。

	original	shrink
実行サイクル数	58,051,566,369	80,822,573
CPI	6.071320	6.186318
2 次データキャッシュミス率	6.8466%/I	6.7249%/I
2 次命令キャッシュミス率	0.0054%/I	0.0719%/I
2 次キャッシュミス率	6.8519%/I	6.7967%/I

表 6 ダイレクトマップ構成に対する評価（メモリアクセスデータ）

	original	shrink
算術命令	24.3994%	17.8008%
データ転送	35.5414%	37.4675%
制御	20.1778%	22.1574%
cc 生成	18.4331%	21.0549%
nop	1.4482%	1.5197%

表 7 命令コンビネーションデータ

4.2 2way 構成の時の評価

2way 構成向けの縮小バイナリの評価は、ソフトウエアシミュレータ上で行った。 configuration は、 2 次キャッシュ構成以外は SPARC64 GP と同様にした。

2way の時のメモリアクセスパターンのデータを表 8に示す。

	original	shrink
実行ステップ数	58,469,055,547	80,537,490
2 次データキャッシュミス率	3.5100%/I	3.5337%/I
2 次命令キャッシュミス率	0.0026%/I	0.0284%/I
2 次キャッシュミス率	3.5126%/I	3.5621%/I

表 8 2way 構成に対する評価（メモリアクセスデータ）

	original	shrink
算術命令	24.3994%	15.6909%
データ転送	35.5414%	35.4887%
制御	20.1778%	24.4881%
cc 生成	18.4331%	23.2201%
nop	1.4482%	1.1131%

表 9 命令コンビネーションデータ

2 次キャッシュミス率の誤差はほぼ 1.4% と、良好な結果が得られた。

また、ダイレクトマップ向けの縮小プログラムを、2way 用の configuration で評価すると、表 10のように誤差が大きくなった。これは、フェーズ B においてキャッシュミスが起きていない部分を縮小したことにより、キャッシュミス率がオリジナルと較べて小さくなったためである。

	original	shrink
実行ステップ数	58,469,055,547	80,750,212
2 次データキャッシュミス率	3.5100%/I	2.6469%/I
2 次命令キャッシュミス率	0.0026%/I	0.0283%/I
2 次キャッシュミス率	3.5126%/I	2.6753%/I

表 10 ダイレクトマップ向けプログラムの 2way 構成に対する評価

5. ま と め

CINT2000 の 181.mcf に対して、オリジナルプログラムのメモリアクセスパターン、命令コンビネーションといった特性を保持しつつ、実行ステップ数を約 80M にできた。これはオリジナルプログラムの 58G に比べて約 700 の 1 である。この縮小プログラムを使用すれば、実行時間が実機の 10,000 倍のシミュレータでも現実的な時間で評価を行える。

181.mcf は、2 つのフェーズ（フェーズ A と B）から成っている。フェーズ A ではプログラム構成が規則正しいため、CFP2000 のプログラムに使用した方法を使用して縮小を行うことができる。しかしながら、フェーズ B では、ループ内で動的に変動するデータ構造に伴って、挙動が大きく変化する。そのため、同じ方針で縮小作業を行うと、オリジナルの特性を保持することができない。そのため、フェーズ B 全体のキャッシュミス回数および実行命令数の情報を採取し、この情報に基づいてループの縮小作業を行った。

ここでフェーズ B のループにおけるキャッシュミスは、2.6MB のデータのラインコンフリクトにより起きていた。このプログラムにおけるラインコンフリクトは、データのメモリ上の配置に起因する。そのためキャッシュサイズが 2MB の時には、ダイレクトマップ構成と 2way 構成でラインコンフリクトの起こり方が異なっていた。従って、2 つの構成に対してそれぞれ別の縮小プログラムを作成する必要があった。

各々の特性に着目して縮小作業を行った結果、ダイレクトマップ構成と 2way 構成向けの縮小プログラムを作成することができ、2 次キャッシュミス率の誤差は 2% 以下にすることができた。

181.mcf は、キャッシュサイズやキャッシュ構成で、大きく挙動が変わるため、ターゲットとなるアーキテクチャに対して縮小作業を変更する必要がある。これらの点を含めて、縮小プログラムの方法論を確立していくことが今後の課題である。

参 考 文 献

1) T. M. Conte, M. A. Hirsch, and K. N. Menezes, “Reducing state loss for effective trace sampling of superscalar processors,” *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 468-477, October 1996. IEEE Comuter Society.
2) P. K. Dubey and R. Nair, “Profile-driven sampled trace generation,” Technical Report RC 20041, IBM Research Division, April 1995.
3) V. S. Iyengar, L. H. Trevillyan, and P. Bose, “Representative traces for processor models with infinite cache,” *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, February 1996.
4) AJ KleinOsowski, John Flynn, Nancy Meares and David J. Lilja, “Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research,” *Workshop on Workload Characterization, International Conference on Computer Design*, Austin, TX, September 18-20, 2000.
5) J. L. Henning, SPEC CPU2000: Measuring CPU Performance in the New Millennium, *IEEE Computer*, 33(7):28-35, July 2000. SPEC CPU2000 Press Rlease FAQ, availale at http://www.spec.org/osg/cpu2000/press/faq.html
6) 稲田由江, 河場基行, 安里彰, “SPEC CFP2000 ベンチマークの縮小プログラム開発手法とその評価,” システム評価研究会, February 14-15, 2002.
7) T. Sherwood and B. Calder, ”Time varying behavior of programs,” Technical Report CS99-630, University of California, August 1999.
8) Andreas Löbel, MCF Version 1.2 – A network simplex implementation, available at http://www.zib.de/Optimization/Software/Mcf/mcf1-2.pdf