最早実行可能条件解析を用いたキャッシュ最適化手法

3H−7

稲石 大祐†, 木村 啓二†, 藤本 謙作† 尾形 航†, 岡本 雅巳‡, 笠原 博徳†
†早稲田大学理工学部電子電気情報工学科, ‡（株）東芝

1 はじめに

単一プロセッサのキャッシュの利用効率向上はシステムの処理速度に大きく影響するため、キャッシュ最適化の研究は広く行なわれている。コンパイラによる代表的なキャッシュ最適化としては、キャッシュプリフェッチング [7] のようにアクセスされると予測されるデータを予めキャッシュに読み込んでおくことによりキャッシュヒット率を高める方法や、ループフュージョン、ループインターチェンジ、ブロッキング等、ループリストラクチャリング [5, 6] によりローカリティを高める手法が挙げられる。しかしこれらのリストラクチャリングは各々のループ内、もしくはフローグラフ上で隣り合ったループに対して行なわれる局所的なものである。そこで本稿では、OSCAR Fortran マルチグレイン並列化コンパイラ [1, 3] における粗粒度タスク間並列性抽出手法である最早実行可能条件解析 [1, 2, 3] を利用した、大域的なコード移動による大域的キャッシュ最適化手法を提案する。

2 最早実行可能条件解析

OSCAR Fortran マルチグレイン並列化コンパイラ [1, 3] における粗粒度並列処理手法 (マクロデータフロー処理) では、Fortran プログラムは次の 3 種の MT [1, 2, 3] に階層的に分割される。

- BPA (Block of Pseudo Assignment statements): 基本ブロック、および複数の小基本ブロックを融合/分割したブロック
- RB (Repetition Block):最外側ナチュラルループ
- SB (Subroutine Block):サブルーチン

分割された MT 間の制御フロー解析、データ依存解析により各階層でのマクロフローグラフ (MFG) が生成される。しかし MFG は MT 間の制御フローとデータ依存を表したものにすぎず、その並列性を示すものではない。従って MT 間の並列性を抽出するには、制御依存とデータ依存を同時に解析する必要がある。そこで、制御依存とデータ依存を考慮した MT 間の最大の並列性を表すものとして、各 MT の最早実行可能条件 [1, 2, 3] を用いる。マクロタスク i(MTi) の最早実行可能条件とは、MTi が最も早い時点で実行可能となるための条件である。ただし、このマクロデータフロー処理における実行可能条件は、次の 2 つの実行条件を仮定して求められる。1)MTi がマクロタスク j(MTj) にデータ依存するならば、MTj の実行が終了するまでは MTi は実行開始出来ない。2)MTj の条件分岐先が確定すれば、MTj の実行が終了しなくても、MTj に制御依存だけしている MTi は実行を開始することが出来る。

MTi の最早実行可能条件の一般形は次の通りである。

```
[ (MTi が制御依存する MTj が MTi を含むパスに分岐する)
  AND
   (MTi がデータ依存する全てのマクロタスク MTk
    (0 ≦ k < |N|) の実行が終了する
     OR
    MTk が実行されないことが確定する)]
```

各 MT の最早実行可能条件を求め、さらに冗長な条件を削除すると、図 1(b) に示すような MTG [1, 2, 3] と呼ばれる無サイクル有向グラフが生成される。MTG において各ノードは MT を表し、実線のエッジはデータ依存を表す。

* A Cache Optimization Scheme Using Earliest Executable Condition Analysis
Daisuke Inaishi†,Keiji Kimura†, Kensaku Fujimoto†, Wataru Ogata†,Masami Okamoto‡, Hironori Kasahara†
†Department of Electrical, Electronics and Computer Engineering, Waseda University ‡TOSHIBA Corporation

3 コード移動によるキャッシュ最適化

本章では、前章で解説した最早実行可能条件を用いて、コードの大域的移動を行ない、キャッシュ利用を最適化する手法について述べる。

3.1 大域的なコード移動

本節ではコードを大域的に移動させる方法について述べる。

前述の最早実行可能条件解析を行なうと、オリジナルプログラム中では後に位置するような MT であっても最早実行可能条件さえ満たされていれば、プログラムのどの位置で実行されようともプログラムの意味は正しく保たれる。例えばあるプログラムが 12 個の MT に分割され (図 1(a))、最早実行可能条件解析の結果図 1(b) のような MTG が生成されたとする。

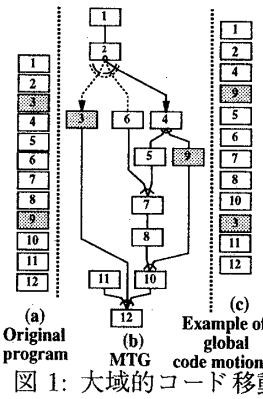

図 1: 大域的コード移動

MT 番号はオリジナルプログラムでの出現順に相当するので、MT9 は MT8 の次に実行されるはずであるが、MTG の情報から MT9 の最早実行可能条件は MT4 の終了 (データ依存) のみであることがわかるため、MT4 の直後に MT9 のコードを移動させることが可能である (図 1(c))。つまり最早実行可能条件を用いた大域的なコード移動は、最早実行可能条件が満たされる範囲内で MT を任意に並べ替えることで実現される。

3.2 キャッシュ最適化

本節ではコード移動によるキャッシュ最適化手法を述べる。

通常あるマクロタスク MTi の実行が終了した時点では、MTi が定義参照したデータがキャッシュ上に存在する確率は極めて高い。従って MTi で定義参照したデータを再び定義参照するような MT を MTi の直後に移動させれば、キャッシュの効率は向上する。図 1において MT4 でアクセスされる配列 A が、MT9 においてもアクセスされるとしよう。オリジナルプログラムの実行順序に従った場合、MT4 実行時にキャッシュ上に読み込まれた配列 A は MT5〜8 を実行する過程でキャッシュから追い出されてしまう可能性がある。しかし MTG 情報から MT9 を MT4 の直後まで移動可能であることがわかるため、MT4 の直後に MT9 を移動させれば MT9 はキャッシュ上の配列 A を利用することができる。このように MT 間で授受されるデータに関する情報を用い、最早実行可能条件が満たされる範囲内で、キャッシュが効率的に利用されるように MT を並べ替える (コードを移動する) ことにより、従来為し得なかったプログラム全体を対象とした大域的なキャッシュ最適化を実現する。

3.3 マクロタスク分割

本節では前節で説明した大域的コード移動によるキャッシュ最適化を効果的にするための前処理について述べる。

第 2 章で述べたようにプログラムは 3 種の MT に分割されるが、プログラムによっては 1 つの MT でキャッシュサイズをはるかに越えるデータを定義参照する場合もある。この場合その MT 内部でキャッシュラインのリプレイスが生じてしまい、大域的コード移動だけではキャッシュの有効利用は難しい。

例えばプログラム中に大量のデータにアクセスする大規模収束ループがある場合を考える。この場合には収束ループの MT をどこに移動させたとしても収束ループ MT 自体を分割しない限りキャッシュヒット率を向上させることは難しいので、収束

ループ MT のボディ部を階層的にサブ MT に分割して MTG を生成し、その MTG を用いて収束ループ内部の MT の並べ替えを行なうことにより実行時間の短縮を図る。

次に複数の MT がキャッシュサイズ以上の共有データをアクセスする場合を考える。例えば MTi と MTj は共有するデータ量が多く、キャッシュサイズ以上のデータにアクセスするループであるとする。この場合 MTi の直後に MTj が実行されるよう並べ替えたとしても、MTi がアクセスするデータ量がキャッシュサイズを越えているため、MTj でアクセスされる共有データが MTi の終了時にキャッシュ上に残っている保証はない (図 2(a))。さらに共有データの一部が残っていたとしても、MTj の実行においてそのデータをアクセスする前に追い出してしまう可能性もある。このような場合には、キャッシュサイズ以上のデータをアクセスする MT をキャッシュサイズ以内のデータをアクセスする複数の MT にループイタレーション方向で分割した後、大域的にコードを移動させれば、キャッシュヒット率の向上が期待できる。図 2(b) のように MTi と MTj を、分割後の MT でアクセスするデータ量がキャッシュに収まるように分割 (MTi^1〜MTi^n,MTj^1〜MTj^n) し、分割後の MTG を用いて MTi^1,MTj^1,MTi^2,MTj^2,...,MTi^n,MTj^n の順序で実行されるように並べ替えれば、キャッシュを効率的に利用することができる。

MT 分割では、MT 間の並列性がコード移動の自由度を決定するため、分割後の MTG で MT 間の依存が増加しないように分割する方が望ましい。また共有データ量が大きい MT 群を分割する場合には、分割後の MTG を用いた MT の並べ替えによってキャッシュの高効率化が望めるように考慮して分割すべきである。このような分割には粗粒度並列処理におけるデータローカライゼーションで用いられる整合分割 [4] が有効である。

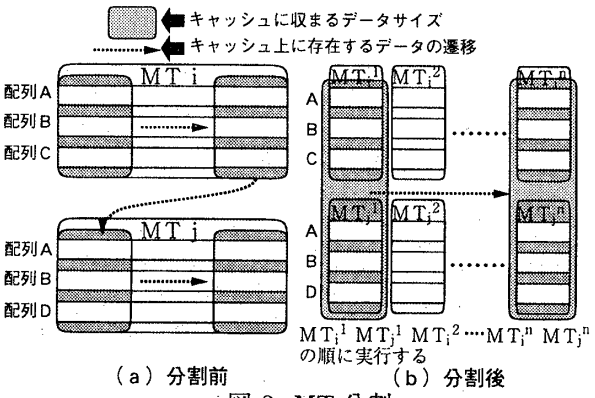

図 2: MT 分割

4 プリプロセッサによる性能評価

本章では本手法の実装方式、その性能評価について述べる。

本手法の実現方法としては、Fortran プログラムを入力とし MT 分割·MT レベルコード移動の結果を逐次型 Fortran プログラムとして出力するプリプロセッサ方式を採用し、OSCAR Fortran マルチグレイン並列化コンパイラを応用することにより実装した。逐次型 Fortran を出力することにより従来のキャッシュ最適化については市販のコンパイラに実装されている最適化を利用することができる。このキャッシュ最適化プリプロセッサを用いて本手法の性能評価を行なった結果について述べる。

性能評価プログラムとして対称帯状係数行列をもつ連立方程式の繰り返し求解法である CG 法 (Conjugate Gradient Method) プログラムと SPEC CFP95 ベンチマークの 1 つである Tomcatv を用いた。オリジナルプログラムとそれをキャッシュ最適化プリプロセッサで最適化したプログラムとを、それぞれコンパイルして実行し、その実行時間を比較した。実行及びコンパイルは Sun Microsystems,Inc.Enterprise3000 (UltraSPARC167MHz,L1cache16KB+16KB,L2cache512KB) 及び富士通 (株)FMV(PentiumII226MHz, L1cache16KB+16KB, L2cache512KB) で行なった。Enterprise3000 の OS は Solaris2.5.1、コンパイラは SPARCompiler FORTRAN 77 4.0、コンパイルオプションには-O5 -native を用いた。FMV の OS は Windows95、コンパイラは DEC DIGITAL Visual Fortran ver5.0A、コンパイルオプションには/optimize:4 を用いた。

CG 法プログラムを用いた性能評価結果を表 1 に示す。FMV では L1cache と L2cache に対して、Enterprise3000 では L2cache に対して最適化を行なった。係数行列サイズは CG 法プログラムで用いられる連立一次方程式の係数行列サイズを、速度向上率はオリジナルプログラムの処理速度に対する最適化後プログラムの処理速度の向上率を表している。同様に Tomcatv を用いた性能評価結果を表 2 に示す。入力ファイルとは評価に用いた Tomcatv の入力ファイル名、実行時間はオリジナルプログラムと最適化後プログラムの実行時間 (単位: 秒) である。この評価結果から、本キャッシュ最適化プリプロセッサにより速度向上が得られることが確かめられた。

表 1 本キャッシュ最適化手法による速度向上 (CG 法)

係数行列サイズ	速度向上率 [%] FMV (L1 Cache)	FMV (L2 Cache)	Enterprise3000 (L2 Cache)
192x192	18.0	13.0	62.3
256x256	24.7	19.7	36.4
384x384	25.0	20.1	31.6
512x512	24.5	21.1	20.2

表 2 本キャッシュ最適化手法による速度向上 (Tomcatv)

入力ファイル	FMV 実行時間 [s] オリジナル	最適化後	速度向上率	Enterprise3000 実行時間 [s] オリジナル	最適化後	速度向上率
train	63.5000	53.7715	18.1%	63.2647	55.8060	13.4%
ref	394.4688	335.8691	17.4%	383.1990	342.4900	11.9%

5 まとめ

本稿では、最早実行可能条件解析を用いた大域的コード移動による単一プロセッサキャッシュ最適化手法を提案した。本手法は OSCAR マルチグレイン並列化コンパイラを用いて、Fortran プログラム入力から最適化された逐次型 Fortran プログラムを生成するプリプロセッサとして実現されている。本プリプロセッサによる性能評価結果から CG 法プログラムで最大 62%(UltraSPARC)、Tomcatv で最大 18%(PentiumII) の速度向上が得られ、本手法の有効性が確かめられた。

今後の課題としては、本手法のマルチプロセッサシステム用キャッシュ利用最適化手法への拡張が挙げられる。

本研究の一部は、通産省次世代情報処理基盤技術開発事業並列分散分野マルチプロセッサコンピューティング領域研究の一環として行なわれた。

参考文献

[1] H.Kasahara, H.Honda, A.Mogi, A.Ogura, K.Fujiwara S.Narita, "A Multi-Grain Compilation Scheme for OSCAR", 4th Intern. Workshop on Languages and Compilers for Parallel Computing, Aug. 1991.
[2] 本多, 岩田, 笠原, "Fortran プログラム粗粒度タスク間の並列性の検出手法", 信学論, J73-D-I(12), Dec. 1991.
[3] 笠原 博徳, "並列処理技術", コロナ社, Jun. 1991.
[4] H.Kasahara, A.Yoshida, "A Data-Localization Compilation Scheme Using Partial-Static Task Assignment", Journal of Parallel Computing ,Vol.24,No.3,pp579-596, May.1998
[5] M.S.Lam, E.E.Rothberg, M.E.Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", ASPLOS IV, Apr.9-11, 1991
[6] M.Wolfe, "High Performance Compilers for Parallel Computing", The Addison-Wesley Publishing Company, 1996
[7] T.C.Mowry, M.S.Lam, A.Gupta,"Design and Evaluation of a Computer Algorithm for Prefetching", ASPLOS V, Oct.1992