社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－ARC－159 (8)
2004／7／30

ハイパースレッディング環境における投機的スレッドを用いたキャッシュ効率化

本田 大[†] 斎藤 史子[†] 山名早人[‡]

概要 近年、CPU処理速度と主記憶からのデータ転送速度との間の格差が顕著になってきているため、キャッシュメモリの重要性が高まってきている。しかし、特に、非線形なアクセスパターンを示すポインタ遷移プログラムでは、キャッシュミスが頻発する。この問題に対し、余剰なCPU資源を利用して、単数あるいは複数のHelperスレッドを実行させ、キャッシュミスレイテンシを隠蔽するPre-Executionが提案されている。本論文では、Pre-Executionにおける、Helperスレッドとメインスレッドとの間の効率的な同期手法を提案する。さらに、従来方式よりprefetchする領域を拡大し、2次キャッシュの効果を高める手法を検討する。Intel Xeonプロセッサでの実機による性能評価の結果、SPEC2000 181.mcfにおいて、平均36.4%の2次キャッシュミスを削減し、3.26%の処理性能高速化を達成できた。

An Efficient Caching Technique Using Speculative Threads on Hyper-Threading Technology

Dai HONDA[†] Fumiko SAITO[†] Hayato YAMANA[‡]

Abstract Recently, the gap between CPU processing speed and the data transmission speed from the main memory has greatly influenced execution speed. Thus data caching technique become more important However, in pointer-based programs which have a nonlinear access pattern a cache memory does not function effectively. To solve this problem, Pre-Execution is a cache miss latency tolerance technique that uses one or more helper threads running in spare CPU's 'resources ahead of the main computation. This paper proposes the synchronous technique of the Helper thread. Furthermore, this paper examines the expanded domain to prefetch in comparison with the conventional Pre-Execution scheme and the technique of heightening the effect of a second level cache is examined. The performance evaluation of SPEC2000 181.mcf on the Intel Xeon processor shows that the average of 36.4% second level cache misses is reduced, and 3.26% of the processing speed is improved.

1. はじめに

近年、CPUとメモリとの間の性能のギャップが年々広がってきている。CPUの処理速度は年率数十%向上しているのに対し、メモリの処理速度は年率数%の向上に留まる。例えば、最近のプロセッサでは、L1キャッシュアクセスに対し、メインメモリへのアクセスのペナルティは、200倍近くある。この差を隠蔽するために、キャッシュメモリが実装されている。その結果、キャッシュメモリを有効活用するための最適化は、性能向上に欠かすことができなくなっている。特に、キャッシュ最適化に関する研究は盛んに行われてきている。以下に、キャッシュ最適化に関する研究状況を示す。

キャッシュ最適化には、コンパイラを利用した研究が多く行われている。コンパイラによるキャッシュ最適化には、ループインターチェンジ、ループ融合、ストリップマイニング、ループ・タイリング、ループ・アンローリングなどのループ変換によりアクセスパターンを変更する手法がある。また、単一ル

† 早稲田大学大学院理工学研究科
Graduate School of Science and Engineering, Waseda University
‡ 早稲田大学理工学部
School of Science and Engineering, Waseda University

ープもしくはループ融合されたループを対象に、イタレーション間での配列内および配列間パディング手法も提案されている[1]。

コンパイラによる最適化を適用しても、ポインタを利用した非線形なアクセスが多いプログラムやRDBMSなどでは、キャッシュミスが多発し、キャッシュメモリの効果があまり得られない。このような、キャッシュミスを頻発するロード命令は、delinquentロードと呼ばれている[2]。delinquentロードによる、ロード転送遅延問題を解決する手法の一つとして、スレッドレベル並列性(Thread Level Parallelism)を利用したPre-Execution[2][3][4][5][6][7]が提案されている。Pre-Executionとは余剰なCPU資源を利用してHelperスレッドを作成し、メインスレッドの実行を補助する技術である。Helperスレッドの実行結果を、適切なタイミングでメインスレッドが使用できるためには、これらの間で同期を取ることが重要になる。

本論文では、Helperスレッドとメインスレッドとの間で効率的な同期処理を行い、メインスレッドの同期処理を削減するPre-Execution手法を提案する。これによって、Pre-Executionでの同期にかかるオーバヘッドを削減することができる。また、Helperスレッドの起動と待機を、メインスレッドが実行したイタレーション数とHelperスレッドが実行したイタレーションの数の差を計算すること（ヒステリシス）で決定する手法を検証した。第2節で従来手法を紹介し、第3節で提案手法、第4節で実験方法および評価結果と考察を示す。第5節で今後の課題を示して、まとめとする。

2. 従来手法

従来のHelperスレッドで実装されたPre-Executionには、静的に実装する手法[2][3][4][5]と、動的に実装する手法[6][7]に大別できる。

静的に実装する手法の特徴は、既存のハードウェアを変更することなく、Pre-Executionを実現し、性能向上を見込むことができることである。文献[2]の手法では、Pre-Execution専用の命令を独自に作り、プログラムソース内に専用命令を挿入することで、Pre-Executionを実装している。文献[3][4]の手法では、コンパイルされた後のバイナリコードからPre-Execution対象のコードを抽出することで、Helperスレッドが実行する命令を決める。文献[2][3][4]では、Helperスレッド生成時に、Helperスレッドの実行に必要なデータをメインスレッドから渡すことで、スレッド間の同期をとっている。よって、スレッド生成とデータコピーのオーバヘッドがかかる。また、文献[5]の手法では、コンパイル前にSource to Sourceによるプログラム変換を行うことでPre-Executionを実装している。文献[5]では、セマフォを使用して、Producer-Consumer方式でスレッド間の同期を実現している。

動的に実装する手法の特徴は、プログラマやコンパイラに頼らず、実行時の情報とハードウェアの状態から、Pre-Executionを実現できることである。文献[6]の手法では、Helperスレッドによって、求めた値をメインスレッドが再利用する手法も提案されている。文献[7]の手法では、Pre-Execution専用ハードウェアを提案し、プログラム実行時に、Pre-Executionすべき箇所の特定とHelperスレッドが実行する命令の生成を行う。文献[6][7]では、スレッド間で共有する専用ハードウェアを仮定し、実行結果を参照する。

しかし、スレッド生成やセマフォを使用するには、オーバヘッドがかかる。また、専用ハードウェアを仮定すると、チップ容量の問題が生じる。

3. 提案手法

第2節で述べた従来手法の問題点に対して、我々は、Helperスレッドとメインスレッドとの間の効率的な同期手法を提案する。これによって、スレッドの生成やスレッド間の同期にかかるオーバヘッドを削減できる。

3.1 Intel SMT アーキテクチャ

IntelのSMT(Simultaneous Multi-Threading)アーキテクチャ対応プロセッサは、Hyper-Threadingを実装している。Hyper-Threadingに対応したCPUでは、メインスレッドとHelperスレッド間で実行資源を共有できる。このように、論理CPUが2つあると仮定で

きるので、1 本のメインスレッドと、1 本 Helper スレッド、つまり合計 2 本のスレッドによる実験を行った。

3.2 プログラムのモデル化

Hyper-Threading 技術を利用し、Helper スレッドをメインスレッドに先行して実行させる。Helper スレッドはメインスレッドの処理中に、後で必要となるデータを prefetch することによって、メインスレッドのキャッシュミスを原因とするストールを隠蔽する。本手法のモデル図を図 1 に示す。

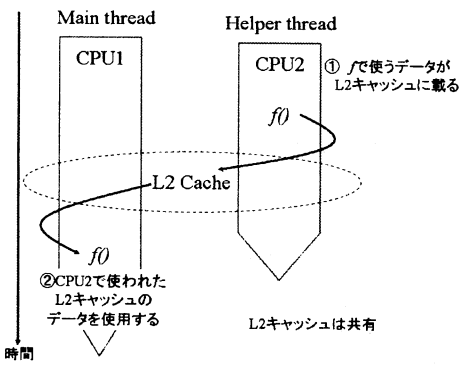

図 1:本手法のモデル図

メインスレッドに対し、Pre-Execution 用の Helper スレッドを 1 本作成する。Helper スレッドはデータをキャッシュに載せるためだけに使用する。そのため、Helper スレッドは、メインスレッドに影響を与えるストア命令を発行しない。これにより、メインスレッドにおける実行の正確性を保証し、かつ、Helper スレッドにおける命令数を削減できる。

3.3 プログラムの実行フロー

本手法では、Pre-Execution の適用対象をプログラムの最内ループとする。メインスレッドが Pre-Execution の適用対象ループに入るまでは、シングルスレッドのみで実行する。Pre-Execution 対象ループに入ってからは、Helper スレッドを作成し、2 スレッドによる並列処理を開始する。アプリケーションの動作を図 2 に示す。

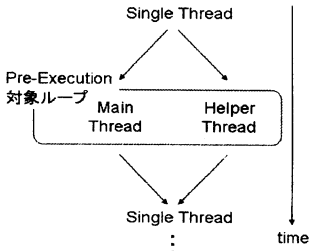

図 2:アプリケーションの実行フロー

3.4 Helper スレッドの生成

Helper スレッドの作成には、WIN32API の CreateThread()を用いて実装した。Helper スレッドが実行する命令は、delinquent ロードと依存関係がある命令のみで構成する。

3.5 スレッド間の同期モデル

Helper スレッドの実行がメインスレッドの実行より、先行しすぎないために、スレッド間で同期をとる必要がある。なぜなら、メインスレッドが必要とするデータを使用する前に、Helper スレッドによって、prefetch しなければならないからである。本手法の同期モデルを図 3 に示す。

```
if ( Thread Distance >= M )
{
    ヒステリシス=
      メインスレッドの実行済みイタレーション数
      −Helperスレッドの実行済みイタレーション数
    if(ヒステリシス >= N)
    {
        Helper スレッド起動
    }
    else
    {
        Helperスレッド待機
    }
}
else
{
    メインスレッドとHelperスレッドを
          並列実行
}
```

図 3:スレッド間の同期手法

図 3 における Thread Distance とは、実行時に動的に変更されるメインスレッドと Helper スレッドのイタレーションの距離を表す。ここでいうイタレーションの距離とは、同一ループ内におけるメインスレッドが実

行中のイタレーションより、Helperスレッドが何イタレーション先行しているかを示している。図3中のMとは、Helperスレッドがメインスレッドに先行しすぎないために設定した静的な値である。また、ヒステリシスとはメインスレッドが実行したイタレーション数とHelperスレッドが実行したイタレーションの数の差である。Nとは Helperスレッドが待機中になった後に、メインスレッドが終了したイタレーションの数を表している。

本手法では、同期をとるためにグローバル変数を使用した。メインスレッドはグローバル変数に、現在まで実行したイタレーションの数を書き込む。グローバル変数の値が、静的に設定した M の値以上になると、Helperスレッドは Pre-Execution を停止し待機する。次に、Helperスレッドは、グローバル変数とHelper スレッドが実行済みであるイタレーション数の差を計算する。この結果と、静的に設定したNの値より、Helperスレッドが待機するか、待機中から Pre-Execution を起動するかを決定する。この手法が有効である理由として、Helperスレッドの命令数がメインスレッドに比べ短く、Helperスレッドが常に先行して実行できるからである。なお、スレッドの待機は、スピンロックで実装している。

4. 実験と評価

本節では、第3節で説明した手法の有効性を検証するための評価環境と実験結果を示す。従来のシングルスレッドによる実行では、キャッシュミスを頻発するアプリケーションに対して、提案手法を適用する。2次キャッシュミス数、およびシングルスレッドとの速度向上比を求めることによって、実行性能を評価する。

4.1 評価環境

実行環境を表1に示す。2次キャッシュミスを測定するツールとして、VTune Performance Analyzer 7.1 を使用した。実行時間の計測には、timeGetTime()関数を使用した。

表1：実行環境

CPU	Xeon 2.4GHz
L1 Data Cache	8KB,4-way-set associative 32-byte Line
L2 Data Cache	512KB,8-way-set associative 64-byte Line
Main Memory	1GB
OS	Windows XP Professional
Compiler	Intel C++ Compiler 8.0.48
Compile Option	/Zi /Zd /QaxN /O3
Link Library	winmm.lib

実行環境における、キャッシュミスにおけるアクセスレイテンシを表2に示す。評価には、Cache Burst[7]を用いた。

表2：キャッシュミスレイテンシ

L1 Cache Access	1 Cycle
L2 Cache Access	18 Cycle
Main Memory Access	213 Cycle

L2 キャッシュへのアクセスレイテンシと比較して、メインメモリへのアクセスは、レイテンシが約 11.8 倍となっている。そこで、メインメモリへのアクセスを減らすことで計算機の速度向上を図る。

4.2 評価対象

実行環境における、SPEC 2000 CINT ベンチマークの2次キャッシュミス率を表3に示す。なお、この評価に関してのみ、OS は Redhat Linux 9 OS(kernel 2.4.19)上での、Valgrind ツール[9]を用いて導出した。なお、ベンチマークの入力は、ref 入力セットを使用した。

表3 :SPEC2000 ベンチマークにおける2次キャッシュミス率

アプリケーション	2次キャッシュミス率
256.bzip2	0.8%
164.gzip	1.7%
176.gcc	0.0%
181.mcf	13.5%
197.parser	0.4%
300.twolf	3.5%
175.vpr	1.7%

表3に示すように、181.mcf が最も2次キャッシュミス率が高い。したがって、評価プ

ログラムとして、181.mcf を選択した。181.mcf は、大規模交通流モデルの最適化プログラムである。

表 4 に、181.mcf に対して提案手法を適用しない場合の L2 キャッシュミス数、および、関数が全体の実行時間に占める比率を示す。

表 4:181.mcf 関数別キャッシュミスの割合

関数名	2 次キャッシュミス数	実行比率
price_out_impl()	2,230,461,558	41%
reflesh_potntial()	2,140,949,394	39%
bea_compute_red_cost()	548,156,880	10%
compute_red_cost()	268,263,348	5%

表 4 で示したキャッシュミス数の大部分は、一部の命令を原因としている。Vtune で解析した結果、181.mcf でキャッシュミスが頻繁に発生する原因は、双方向リストにおけるポインタの参照先アドレスを対象としたロード命令であることが判った。181.mcf における、delinquent ロードの例を図 4 に示す。図 4 では、表 4 に示した最も 2 次キャッシュミス数が多い関数を例としてとりあげた。

```
Long price_out_impl()
{
    for(i=0;i<MAX;i++)
    {
        while( arcin )
        {
            tail = arcin->tail;
            if(tail->time + arcin->org_cost < latest)
            {                                    ↖
                arcin = tail->mark;              Cache Miss!!
            }
            ...
        }
    }
}
```

図 4:delinquent ロード位置

Pre-Execution の実装は、2 次キャッシュミス数が最も多い price_out_impl()関数に対して適用した。さらに price_out_impl()関数内で呼び出される compute_red_cost()関数に対しても適用した。

4.3 評価結果

同期変数 M の値とヒステリシス N の値を変化させた場合の、Pre-Execution を適用した場合と、適用していない場合と速度向上率を図 5 に示す。

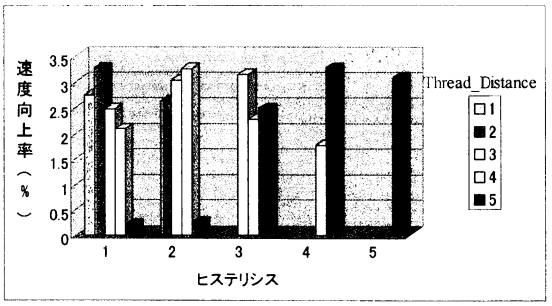

図 5:速度向上率

図 5 に示すように、同期変数 M の値が 4、ヒステリシス N の値が 2 のとき、最も高い速度向上率が得られた。本実験では、Helper スレッドの方がメインスレッドに比べ、実行コードが短いので、図 5 に示すように、ヒステリシスが有効に機能していることがわかる。同期変数の値に比べ、ヒステリシスの値が小さい場合、Helper スレッドが prefetch したデータが 2 次キャッシュからはずれてしまい、Pre-Execution の効果が得られなかったと考えられる。

実行速度が最速のときの 2 次キャッシュミス数を表 5 に示す。表 5 に示すように、どちらの関数においても、キャッシュミスを削減できている。

この結果は、1 イタレーション毎に Helper スレッドを作成する手法[3]と比較して、9.11%高い速度向上率である.

表 5:本手法の実験結果

関数名	提案手法適用前のキャッシュミス数	提案手法適用後のキャッシュミス数	2 次キャッシュミス削減率	Helper スレッドの命令数削減率
price_out_impl()	2,230,461,558	1,489,501,953	33.2%	56.2%
compute_red_cost()	268,263,348	162,418,341	39.5%	40%

5. おわりに

本稿では、Helperスレッドで同期を行い、かつ適用範囲を拡大した Pre-Execution 手法を Intel Xeon プロセッサ上で検証した。結果、2 次キャッシュミス数を平均 36.4%削減し、実行時間を 3.26%短縮することができた。

今後は、より多くのアプリケーションに適用し、効果を確認する予定である。また、スレッド間のオーバヘッドを軽減するためにも、新しい同期法の考案が必要だと考える。さらに、現在の Hyper Threading 技術を拡張し、論理 CPU を増やすことで、割り当てる Helperスレッドの数も CPU 台数に対応させ、Pre-Execution の効果をより高めていきたい。

謝辞

本研究の一部は、日本学術振興会 21 世紀 COE プログラム「プロダクティブ ICT アカデミア」の支援により行われた。

参考文献

[1]石坂 一久, 小幡 元樹, 笠原 博徳,"配列間パディングを用いた粗粒度タスク間キャッシュ最適化",情報処理学会論文誌, Vol. 45, No. 4 ,pp.1063-1076, April, 2004

[2]C. K. Luk:"Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", In Proc. of 28th Intl. Symp. on Computer Architecture(ISCA) ,pp.40-51,2001

[3]J.Collins,H.Wang,D.Tullsen,C.Hughes,Y-F.Lee , D.Lavery , J.Shen : "Speculative Precomputation : Long-range Prefetching of Delinquent Loads",In Proc. of 28th Intl. Symp. on Computer Architecture(ISCA) ,pp. 14-25, 2001

[4]Steve S.Liao,Perry H.Wang,Hong Wang , Gerolf Hoflehner , Daniel Lavery , John P.Shen:"Post-Pass Binary Adaptation for Software-Based Speculative Precomputation" , In Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI) ,pp.117-128, 2002

[5]Dongkeun Kim,Donald Yeung:"Design and Evaluation of Compiler Algorithms for Pre-Execution",In Proc. of 9th Intl. Conference on Architectual Support for Programming Languages and Operating System(ASPLOS) ,pp.62-73,2002

[6]Amir Roth and Gurindear S.Sohi:"Speculative Data-Driven Multithreading ",In Proc. of 7th Intl. Symp. on High Performance Compute Architecture(HPCA), pp.37-48,2001

[7]C.Zilles,G.Sohi:"Execution-based prediction using speculative slices", In Proc. of 28th Intl. Symp. on Computer Architecture (ISCA) ,pp.2-13,2001

[8]Valgrind http://developer.kde.org/~sewardj/docs-2.0.0/manual.html

[9]Cache Burst 32 http://user.rol.ru/~dxover/cburst/