キャッシュ・ミス頻発命令を考慮したメモリ・システムの高性能化

三輪, 英樹
九州大学

堂後, 靖博
福岡大学

グラール フェヘイラ, ヴィクトル マウロ
九州大学

井上, 弘士
九州大学

他

出版情報：デザインガイア2004, 情報処理学会研究報告2004-ARC-160, pp.107-112, 2004-12. 情報処理学会ARC研究会
バージョン：

キャッシュ・ミス頻発命令を考慮したメモリ・システムの高性能化

三輪英樹[†] 堂後靖博[††] ヴィクトル M. グラール フェヘイラ[†]
井上弘士[†] 村上和彰[†]

マイクロプロセッサと主記憶との動作周波数差は，年々拡大する一方である．両者の周波数差は，マイクロプロセッサの性能阻害要因であり，一般的にメモリ・ウォール問題と呼ばれる．本稿では，メモリ・ウォール問題の解決策のうち，再計算に基づくメモリ・システム高性能化手法 CCC (Computing Centric Computation) を提案する．CCC は，キャッシュ・ミス頻発命令を実行する代わりに再計算を行なうことで，主記憶へのアクセス回数を削減する．本稿では，性能向上が得られる可能性があるかどうかに関する予備的な評価を行なった．評価対象ベンチマークにおいて最大 45.3% の実行サイクル数削減率を達成した．

A Method for Improving Memory System Performance Exploring Delinquent Loads

Hideki MIWA,[†] Yasuhiro DOUGO,[††]
Victor M. GOULART FERREIRA,[†] Koji INOUE[†]
and Kazuaki MURAKAMI[†]

In recent years, the performance gap between microprocessor speed and main memory latency has been increasing. This problem prevents higher throughput improvements and is well-known in the literature as the Memory-Wall Problem (MWP). This paper proposes a new method to minimize the MWP effect by means of [re-computation]. The basic idea is to replace frequently cache-missed loads (or delinquent loads) with a piece of code that regenerates the missed value (recomputation code). This method can be used to reduce the number of main memory accesses and consequently alliviate the MWP. From the experiments, one can obtain up to 45.3% reduction on computation time for SPEC2000 benchmark programs.

1. はじめに

マイクロプロセッサの動作周波数は年率 55% で向上しているのに対し，主記憶として利用される DRAM (Dynamic Random Access Memory) の場合は年率 7% という低い伸びにとどまっている[5]．両者の動作周波数差は年々拡大する一方であり，マイクロプロセッサのスループット向上を阻害する 1 つの要因となっている．この問題は一般的にメモリ・ウォール問題と呼ばれる．

メモリ・ウォール問題を解決するためには，主記憶へのアクセスレイテンシを低減もしくは隠蔽しなければならない．代表的なアクセスレイテンシ低減手法としてはキャッシュメモリの搭載が挙げられ，多くの商用マイクロプロセッサで実用化されている．一方，主記憶へアクセスする際のレイテンシの隠蔽手法としてはプリフェッチが挙げられ，これまで多くの研究開発が行なわれてきた．特に，Delinquent ロード命令 (以下，DL 命令) に着目したプリフェッチ手法が近年注目されている[4][6]．DL 命令は，キャッシュ・ミスを頻発させる静的なロード命令であり，キャッシュ・ミスの 90% を占める場合もあると報告されている[7]．DL 命令を対象としたプリフェッチは，DL 命令専用アドレス計算コードを別スレッドとして実行し，より正確なアドレスを求めることでプリフェッチ効率を高められる．しかしながら，プリフェッチは主記憶へのアクセスを先見的に行なう手法であり，キャッシュ・ヒット率は向上するものの主記憶へのアクセス回数は削減できない．加えて，投機的なアドレス計算に基づくため，誤ったアドレスのデータをフェッチすることで必要なデータをキャッシュ・メモリから追い出し，ヒット率の低下を招く可能性もある．

本稿ではメモリ・ウォール問題への対策として，再計

† 九州大学 Kyushu University
Email: arch-ccc-lpc@c.csce.kyush-u.ac.jp
†† 福岡大学 Fukuoka University

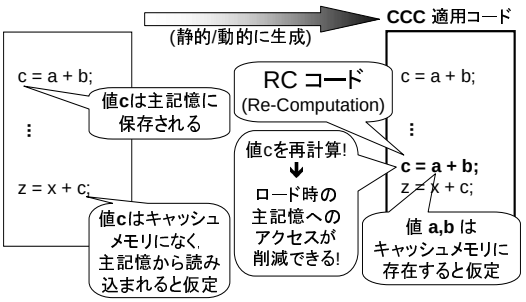

図 1 CCC の概念図

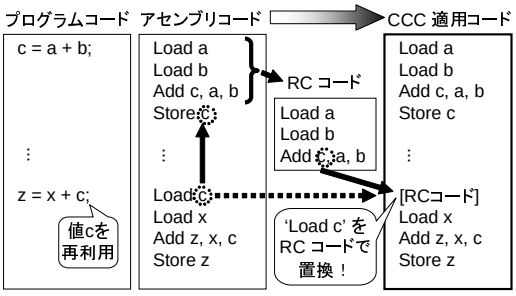

図 2 RC コード生成方法

算に基づくメモリ・システム高性能化手法 CCC (Computing Centric Computaion) を提案する．CCC は，DL 命令を実行する代わりに，計算によりロードすべきデータを求める．チップ外に追い出されたデータをロードするのではなく，チップ内にて再計算することで主記憶へのアクセスそのものを削除する．本稿では，CCC の提案および予備的な評価を行ない，実行サイクル数の削減が可能であるかどうか検討する．なお，本稿では L2 キャッシュ・メモリまでがオンチップ化されている場合を前提とし，キャッシュ・ミスとは L2 キャッシュでのミスを指す．

本稿は以下のような構成である．第 2 節にて提案手法を述べる．次に，第 3 節にて提案手法の予備評価を行ない，その結果について考察する．最後に，第 4 節にて本稿をまとめる．

2. 再計算に基づくメモリ・システム高性能化手法

本節では，再計算に基づくメモリ・システム高性能化手法 CCC を提案し，その概要を述べる．

2.1 CCC の概要

主記憶へのアクセスが必要となる場合，再計算により対象データを求めることができれば，実行サイクル数を削減できる可能性がある．我々は，この仮説を証明するために，再計算に基づくメモリ・システム高性能化手法 CCC を提案する．

CCC の概要を図 1 に示す例を利用して説明する．図の左側が CCC 適用前のプログラムコードであり，右側が CCC 適用後のプログラムコードである．CCC 適用前のコードでは，値 c が コードの前半で定義され，後半で利用される．後半部では，値 c はキャッシュ・メモリに存在せず，主記憶からロードする必要がある場合を仮定する．CCC 適用後のプログラムコードには，値 c を再定義する計算式が追加されている．データを再計算するためのコードを Re-Computation (RC) コードと呼ぶ．

CCC 適用後，RC コードにより値 c を再定義する際に値 a,b がキャッシュ・メモリに存在する場合，値 c はキャッシュ・ミスを発生させることなく再計算できる．CCC の適用により，主記憶へのアクセスを再計算に置換でき，かつその実行時間が主記憶へのアクセス・レイテンシよりも短い場合，実行サイクル数を削減することができる．

2.2 RC コードの生成方法

本節では，RC コードの生成方法を説明する． 図 2 は，第 2.1 節で示した例と同様である．図 2 において，左から 2 番目の枠内が CCC 適用前のアセンブリコードである．

RC コードを作成する場合，まず，ロード対象データの絶対アドレスを調べる．上記の例では，`Load c` 命令におけるロード対象データの絶対アドレスである．次に，ロード対象データの絶対アドレスに対して書き込みを行なった命令を後方から検索する．上記の例では `Store c` 命令がこれに相当する．続いて，アセンブリコード上でレジスタ依存関係のある命令を順に抜き出すことにより，RC コードが生成できる．依存関係の追跡は，ロード命令が出現した時点で終了する．

上記の例における RC コードは，2 個のロード命令および 1 個の加算命令により構成される．一般的には，RC コードに分岐命令およびジャンプ命令が含まれる場合もある (図 3 参照)．このような RC コードでは，複数の実行パスが存在する．

2.3 CCC の実現方法の分類

CCC の実現方法を表 1 に示す．それぞれの概要を以下で述べる．

- **コンパイル時 CCC**: コンパイル時に RC コードを生成し，DL 命令を RC コードで置換する．
- **静的/動的マルチスレッド CCC**: 静的の場合はコンパイル時に，動的の場合は実行時に，それぞれ RC コードを生成する．DL 命令実行時には，

表 1 CCC の実現方法の分類

RC コード生成方法 ＼ RC コード実行環境	同一スレッド	別スレッド
実行前	コンパイル時 CCC	静的マルチスレッド CCC
実行時	実行時 CCC	動的マルチスレッド CCC

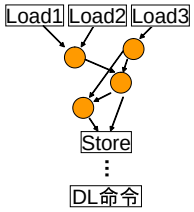

図 3 一般的な RC コードの DFG

通常のメモリ・システムへのアクセスと並行して，RC コードが別スレッドで実行される．

- **実行時 CCC**: 実行時に RC コードの生成を行ない，キャッシュ・ミス発生時に RC コードを実行する．

2.4 CCC の性能向上条件

本節では，第 2.3 節で述べた CCC の実装方法に関係なく，CCC により実行サイクル数を削減するために満たすべき条件について議論する．

CCC により実行サイクル数を削減するためには，多くの場合に以下の条件が満たされる必要がある．

(1) DL 命令がキャッシュ・ミスを発生させること．
(2) DL 命令が RC コードで置換できること．
(3) RC コードの実行サイクル数が，主記憶へのアクセスサイクル数より少なくなること．
(4) RC コード内に含まれるロード命令が，常にキャッシュ・ヒットすること．

まず，(1) および (2) は，CCC を適用できる機会に関する条件である．適用機会が多いほど，効果が表れやすい．次に，(3) および (4) は，CCC によるオーバーヘッドに関する条件である．再計算に要する時間が短く，かつ RC コード中のロード命令がキャッシュ・ヒットしなければ CCC を適用しても実行サイクル数は削減できない．CCC を適用することにより実行サイクル数を削減するためには，アプリケーションが少なくとも上記の 4 条件を満たしていなければならない．

3. 予 備 評 価

本節では，ベンチマークプログラムを利用し，CCC を定量的に評価する．ただし，本節では第 2.4 節で挙げた CCC の実現方法に関係なく，CCC による最大性能向上を評価する．

表 2 シミュレータのプロセッサ構成に関するパラメータ

命令発行方式	アウトオブオーダ
分岐予測器	
種類	2 レベル (gshare, 2K エントリ)
BTB サイズ	512 エントリ, 4 ウェイ
RAS	32
命令デコード幅	8 命令/サイクル
命令発行幅	8 命令/サイクル
IFQ サイズ	8 エントリ
RUU サイズ	64 エントリ
LSQ サイズ	32 エントリ
キャッシュ・メモリ	
L1 データ	32KB (64B/エントリ, 2 ウェイ, 256 エントリ)
L1 命令	32KB (64B/エントリ, 1 ウェイ, 512 エントリ)
L2 共有	2MB (64B/エントリ, 4 ウェイ, 8192 エントリ)
ヒット・レイテンシ	
L1 キャッシュ	1 クロック
L2 キャッシュ	16 クロック
主記憶	250 クロック
メモリ・バンド幅	8B
メモリ・ポート数	2
ITLB, DTLB	
エントリ数	1M entries (4KB/エントリ, 256 エントリ/ウェイ, 4 ウェイ)
ミスペナルティ	30 サイクル
整数演算器 (装置数, 実行レイテンシ, 発行レイテンシ)	
ALU	4, 1 クロック, 1 クロック
乗算器	1, 3 クロック, 1 クロック
除算器	1, 20 クロック, 19 クロック
浮動小数点演算器 (装置数, 実行レイテンシ, 発行レイテンシ)	
ALU	4, 2 クロック, 1 クロック
乗算器	1, 4 クロック, 1 クロック
除算器	1, 12 クロック, 12 クロック
開閉演算器	1, 24 クロック, 24 クロック

3.1 実 験 環 境

CCC の評価を行なうために，マイクロプロセッサシミュレータである SimpleScalar を利用した[2]．シミュレータのプロセッサ構成に関するパラメータを表 2 に示す．

評価対象アプリケーションは，SPEC CPU 2000 ベンチマークセットより以下に示す 11 種類を利用した[3]．これらのアプリケーションは，ミシガン工科大学で開発された MIRV C コンパイラを利用してコンパイルされている[1]．

- 浮動小数点演算プログラム (4 種類):
  177.mesa, 179.art, 183.equake, 188.ammp

- 整数演算プログラム (7 種類):
  164.gzip, 175.vpr, 176.gcc, 181.mcf,
  197.parser, 255.vortex, 256.bzip2

入力データは，SPEC CPU 2000 の Reference 入力を利用した．各ベンチマークの引数は，DL 命令による L2 データ・キャッシュ・ミス回数が最も多くなった設定を採用している．本実験では，シミュレーション時間を短縮するため，プログラム実行開始より 20 億命令実行後の 2 億命令を評価対象とした．

3.2 評価における前提

本評価における仮定は，以下の通りである．

- 本評価において DL 命令は，最もキャッシュ・ミスを頻発させる 16 アドレス分のロード命令と定義する．このように定義する根拠は，16 アドレス分のロード命令により全体のキャッシュ・ミス回数の多くを占めるためである．図 4 は，キャッシュ・ミス回数で DL 命令をランク分けし，1 位，2 位，3 位，および 4 位から 16 位の命令によるキャッシュ・ミス回数の割合を示すグラフである．図 4 より，多くのベンチマークで上位 3 命令により 40% 以上のキャッシュ・ミスを占め，上位 16 命令により 60 % 以上を占めることが分かる．
- 本来, RC コードには，ロード対象データの再計算に必要な全てのパスが含まれるべきである．しかしながら，本評価では，事前に取得したトレースデータから 1 つの実行パスのみ抽出し，RC コードを生成する．
- RC コードに含まれる命令数を評価するためには，全ての RC コードを生成しなければならない．しかしながら，本評価では，RC コードの生成時間を短縮するため，類似した RC コードをグループ化し，各グループごとに 1 個の代表 RC コードを生成する．グループ内の全ての RC コードの命令数を，代表 RC コードの命令数で代用する．
- RC コード中のロード命令は全て L1 データキャッシュにヒットし，第 2.4 節の条件 (4) が常に成立しているものとする．
- RC コードの実行サイクル数が主記憶へのアクセスレイテンシよりも長くなる場合，RC コードによる再計算は行なわず，従来方式と同様にオフチップアクセスを行なう．
- RC コードの実行にともなう命令キャッシュおよびデータキャッシュへの影響については考慮しない．

3.3 評 価 指 標

本評価における評価指標として，CCC を適用した場合の実行サイクル数削減率を採用する．本評価では，

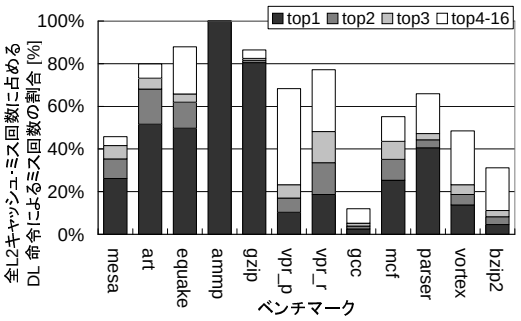

図 4 DL 命令による L2 キャッシュ・ミス回数の割合

各ベンチマークにおいて，CCC を適用した場合の実行サイクル数を以下の式 (1) により近似する．

$$C_{ccc} = C_{ideal} + C_{rc} + C_{norc} \quad (1)$$

- C_{ccc}: CCC 適用後の実行サイクル数.
- C_{ideal}: DL 命令の L2 データ・キャッシュ・ミス・ペナルティを 0 とした場合の実行サイクル数.
- C_{rc}: RC コードで置換可能な DL 命令に関して，ロード対象データの再計算に要する実行サイクル数.
- C_{norc}: DL 命令のうち，RC コードで置換できなかったロード命令の実行サイクル数.

C_{rc} は，以下の式 (2) で近似する．

$$C_{rc} = CPI \cdot \sum_{i}^{dldpc} \sum_{k}^{rccg} I_{rccg}(i,k) \cdot N_{rccg}(i,k) \quad (2)$$

- CPI: 1 命令の実行に要する平均クロックサイクル数.
- $I_{rccg}(i,k)$: アドレスが i の DL 命令のあるインスタンスに対する RC コードのうち，RC コードグループ k の命令数.
- $N_{rccg}(i,k)$: アドレスが i の DL 命令のあるインスタンスに対する RC コードのうち，RC コードグループ k における RC コードの実行回数.

RC コードで置換できなかったロード命令の総実行サイクル数 C_{norc} は，以下の式 (3) で定義する．

$$C_{norc} = (C_{org} - C_{ideal})/N_{dl2miss} \cdot N_{norc} \quad (3)$$

- C_{org}: CCC 適用前実行サイクル数.
- $N_{dl2miss}$: DL 命令による全 L2 ミス回数.
- N_{norc}: RC コードで置換できない DL 命令の総実行回数.

実行サイクル数削減率は，式 (1), 式 (2) および式 (3) より，以下の式により求めることができる．

$$(C_{org} - C_{ccc})/C_{org} * 100[\%]$$

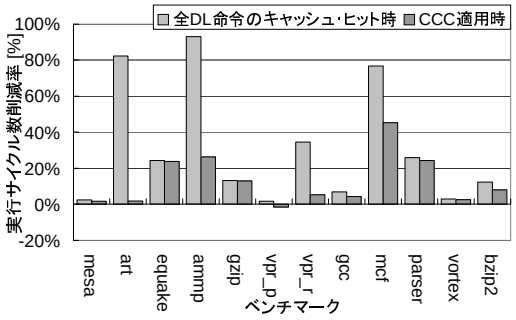

図 5 全 DL 命令キャッシュ・ヒット時および CCC 適用時の実行サイクル数削減率

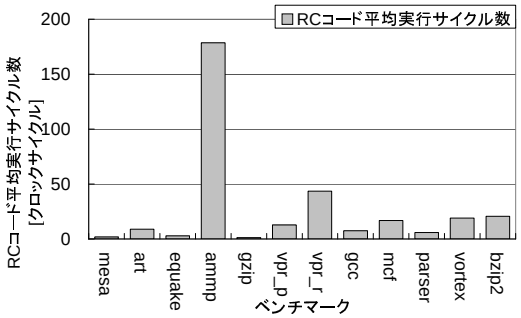

図 6 RC コード平均実行サイクル数

CCC を適用した場合の実行サイクル数削減率を求めるため，実験にて上記 C_{org}，C_{ideal}, $I_{rccg}(i,k)$, $N_{rccg}(i,k)$, N_{norc}, $N_{dl2miss}$, および CPI を求める．

3.4 本評価における RC コードの生成方法

RC コードを生成するためには，コントロールフローおよびデータフローを明らかにする必要がある．本評価では，トレースデータを利用し，1 パスのみのコントロールフローを抽出する．データフローは，アセンブリコードにおいてレジスタの依存関係を追跡するプログラムを利用して抽出する．以上のようにして得たコントロールフローおよびデータフローを利用して，RC コードを生成する．本評価では，DL 命令のあるインスタンスに対する RC コードのみを生成し，$I_{rccg}(i,k)$ を求める．$I_{rccg}(i,k)$ の値に応じて，各 DL 命令が RC コードで置換できるかどうかを分類し，$N_{rccg}(i,k)$ および N_{norc} を求める．

3.5 評価結果および考察

CCC 適用時の実行サイクル数削減率を図 5 に，RC コード平均実行サイクル数を図 6 に，および RC コードで置換できた DL 命令の割合を図 7 に示す．

図 5 より，いくつかのベンチマークでは実行サイクル数を最大で 45.3% 削減できるのに対し，mesa, art, vpr, gcc および vortex ではほとんど効果が得られていないことが分かる．この原因を RC コードの平均実行サイクル数および RC コードで置換できた DL 命令の割合を基に考察する．

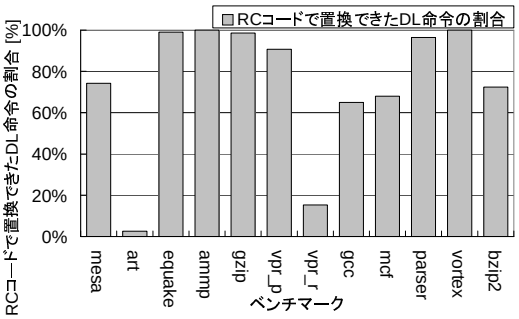

図 7 RC コードで置換できた DL 命令の割合

まず，mesa, vpr_p, gcc および vortex は，図 5 より全 DL 命令のキャッシュ・ヒットを仮定した場合の実行サイクル数削減率が低い．この理由は，第 2.4 節にて述べた性能向上条件 (1) が満たされる場合が少ないためである．このようなアプリケーションでは，DL 命令の影響が小さいため，CCC の効果はほとんど得られない．次に，art および vpr_r については，図 6 および図 7 より RC コード実行サイクル数は少ないものの，RC コードで置換できた DL 命令数が少ないことが分かる．第 2.4 節にて述べた性能向上条件 (2) が満たされない場合が多く，適用できる機会が少ないため十分な効果が得られなかったと考える．最後に，ammp について追加考察する．図 5 では全 DL 命令のキャッシュ・ヒットを仮定した場合の実行サイクル削減可能性は非常に高いが，CCC 適用時にはそれほど効果が出ていない．この理由は，図 6 より，RC コードの実行サイクル数が大きく，第 2.4 節にて述べた性能向上条件 (3) が満たされていないためである．

以上より，第 2.4 節において指摘した性能向上条件のいずれかが満たされない場合，十分な実行サイクル数削減効果が得られないことが確認された．本評価におけるベンチマークの中では，mcf はいずれの性能向上条件も満たしており，大きな実行サイクル数削減効果が表れている．

4. おわりに

本稿では，メモリ・ウォール問題への対策として，再計算に基づくメモリ・システム高性能化手法 CCC を提案した．本稿における予備評価結果より，CCC を実現することで実行サイクル数を削減することがで

きる可能性があることを示した．

今後は，以下の点を中心に研究を進める予定である．

- RC コード生成方法の確立．
- RC コード内のロード命令のキャッシュ・ヒット状況の調査．
- CCC の適用によるキャッシュ・ヒット状況の変化の調査．

謝辞 本研究を進めるにあたり，多くのご指導を頂いた安浦寛人教授をはじめとする研究室諸氏，富士通株式会社の池田正幸氏および丸山拓巳氏，富士通研究所の勝野昭氏および坂本真理子氏に深く感謝致します．本研究は，一部文部省科学研究費補助金 (課題番号: 14GS0218, 14702064) に依る．

参 考 文 献

1) *The MIRV Compiler*. http://www.eecs.umich.edu/mirv/.
2) *SimpleScalar LLC*. http://www.simplescalar.com/.
3) *SPEC - Standard Performance Evaluation Corporation*. http://www.spec.org/.
4) Collins, J. D. et al.: Speculative Precomputation: Long-range Prefetching of Delinquent Loads, *ISCA '01 Proceedings*, pp. 14–25 (2001).
5) Hennessy, J. L.and Patterson, D. A.: *Computer Architecture: A Quantitative Approach 3rd Edition*, Morgan Kaufmann Publishers (2003).
6) Roth, A. and Sohi, G.: Speculative Data-Driven Multithreading, *HPCA '01 Proceedings*, pp. 37–49 (2001).
7) 堂後靖博ほか: キャッシュ・ミス頻発命令とその特徴解析, 情報処理学会研究報告 ARC-160-015 (2004).