2003－HPC－96　(7)
2003／10／16

値再利用機構におけるバッファ置換アルゴリズムの改良

吉川 岳流 †　　梅谷 征雄 ‡
† 静岡大学大学院情報学研究科　　‡ 静岡大学情報学部情報科学科

値局所性を利用した主要な高速化技法の 1 つとして値再利用機構がある。本研究では、この値再利用機構の中心となる、入出力値を保持しておく Reuse Buffer の有効な利用法を模索するものである。今回我々は、2 つのアイデアを ReuseBuffer に適用し、その効果を検討した。1 つは、1 つの命令に対して複数の入出力値の組を保持する方法で、もう 1 つは、バッファのエントリ置換にエントリの再利用率と古さを加味するものである。その結果、前者の手法で効果が見られ、SPEC95,2000 のいくつかのベンチマークにおいて、1 つの命令に対して 1 エントリが対応で置換アルゴリズムが FIFO の場合(再利用機構のない場合に対し 1.3～7.4%向上)よりも、更に+1%前後の速度向上がみられた。

Improvement of the buffer replacement algorithm in value reuse mechanism

Takeru Kikkawa †　　Yukio Umetani ‡
† Graduate schoolof information, Shizuoka Univ.　　‡ Faculty of Information, Shizuoka Univ.

Value reuse mechanism is one of the main techniques using value locality property. In this research, we seek for the effective management of Reuse Buffer(RB) that is the principal part of a reuse mechanism. We applied two ideas to RB and examined thire effects. One is the method of holding a group of two or more input and output value pairs for one instruction, and the other is the entry replacement considering reuse rate and oldness for each entry. As a result, they achieved further 1% processor speedup in some benchmarks of SPEC 95 and 2000 to the conventional case (one entry par to one instruction with FIFO replacement) that improved 1.3 to 7.4% to the case without reuse mechanism.

1. 背景と目的

近年、プロセッサの命令実行の分野において、値局所性(Value Locality)という概念が提案され[2]、これにもとづいてさまざまな研究がなされてきた。[2]において値局所性とは、(静的な)命令ごとに特定の値の出現する傾向のことであるとされている。現在、値局所性を利用した実行時間の短縮化技法には、投機的なもの(値予測に基づく投機的実行)と、そうでないもの(値再利用)がある。

1.1 値予測に基づく投機的実行

これは、各命令について、実行時に出現した値をバッファに保持しておき、それまでの実行で出現した値から実行結果を予測し、それを仮の結果として投機的に処理を進める方法である。命令の入力オペランドが確定する前に実行を開始することができる。結果はあくまで予測であるため、予測がはずれたときの回復の機構が必要であり、はずれたときには回復のために余分なサイクルを消費することになる。値予測に基づく投機的実行に関しては、さまざまな研究がなされており[3]、値予測機構に関しては主なものとして、Last-Value 値予測機構[2][4]、ストライド値予測機構[6]、context-base 値予測機構[5]、2 レベル値予測機構[6]などが提案されている。

1.2 値再利用

もう一方の予測によらない手法のほうは、値再利用と呼ばれている。こちらは、値予測と同様に結果をバッファリングし、それまでの実行で出現した入力値とマッチした場合に、その過去の結果を利用するものである。入力オペランドを比較する必要があるため、前者よりも実際に実行しなければならないステージが多い。また、ループカウンタの増減やそれに伴うアドレス計算のように、出現値が特定の傾向で変化する場合は前者のほうが有利である。

なお、再利用を基本ブロック単位[7][9]や関数単位で行う手法[10]も提案されており、これらの場合、1 度に複数の命令の実行を省略することが可能になる。さらに、再利用単位をコンパイラが指示する手法も提案されている[8][9]が、コンパイラに頼る場合は既存のバイナリプログラムの高速化は図れない。

ここまで値予測に基づく投機実行と値再利用について述べてきたが、この 2 つを併用することも考えられ、実際に Wu らが検討を行っている[11]。

1.3 本研究の目的

本研究では、値再利用機構の ReuseBuffer をより有効に活用する方法を提案し、それによる性能向上を調べた。Reuse Buffer の置換アルゴリズムに関しては研究がなされておらず、性能向上の余

地が残されていると考えられる。

前述のように、値再利用機構に関する研究は、ある程度の命令列単位での再利用や、コンパイラの支援を利用したものなどが行われている。本研究では、再利用の単位は 1 つ 1 つの命令とする。これは構造が簡単なこと、また値再利用機構が提案された[1]においては全体的に性能がよかったのが 1 命令単位での再利用であったことによる。また、再利用機構はハードウェアのみで実現することとする。これは既存のバイナリプログラムにおいても効果が得られることと、再利用単位が 1 命令の場合はソフトウェアでサポートすべき事象は今のところ思い当たらないためである。

本研究では次の 2 点の手法を試みる。

(a) 1 命令に対応する RB のエントリを複数保持

(b) 再利用度、生存期間に基づいて算出した優先度によるエントリ置換

(a)によって、直前に出現した値だけでなく、過去の何種類かの値を再利用することが期待できる。なお、フォローする範囲の点からすると、値予測に基づく投機的実行の、last-value 値予測器(最後に実行されたときの値を予測値とする)に対する、context-base 値予測器(値の出現パターンから値を予測)との対比に似ている。

(b)は、再利用されそうにないエントリの、より早期での入れ替えを目指す。これにより、従来の手法(FIFO)の場合の再利用性の低いエントリであっても掃き出されるのに全エントリ数と同じ回数のエントリ置換を必要とする点を解消することが期待できる。

本研究では Sodani らが提案した値再利用機構[1]を基準とし、改良を加えることによって得られた性能向上を評価する。

2. 基本アーキテクチャ

ここでは値再利用機構を組み込んだアーキテクチャについて述べる。Sodani らの研究[1]、本研究とも、基本構成はここで述べるものに順ずる。

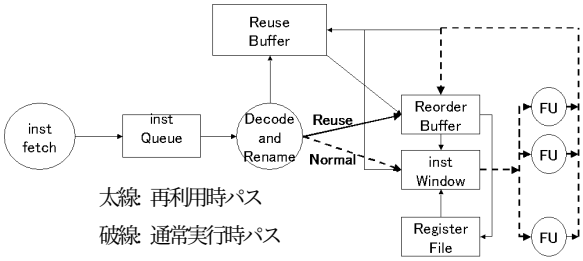
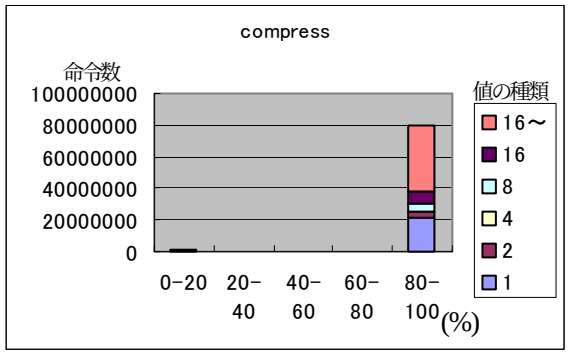


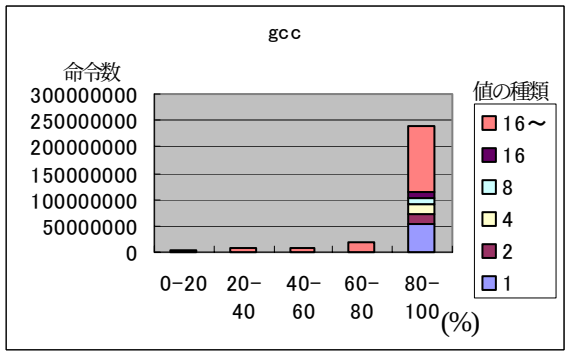
図 1 Microarchitecture with RB

図 1 は、演算結果を格納しておく Reuse Buffer(RB)を組み込んだプロセッサの概要である。デコードステージで RB が命令のアドレスによってエントリが索引され、エントリの内容が有効か、またオペランドが合致しているかがテストされる。テストの結果、再利用可能であれば格納されている演算結果が直接リオーダバッファに送られ、そうでない場合は新規登録用のエントリが予約され実際に演算が行われ、実行完了時に結果が格納される。

3. 準備実験

出現値の振る舞いを理解するために、SimpleScalar3.0[12]の命令レベルシミュレータを使用し、各静的命令について、出現した入力値と値ごとの出現回数を調べた。ここではまだ値再利用は行わない。ベンチマークには、SimpleScalar LLC[12]で提供されている SPEC95 のコンパイル済みバイナリの中から compress、gcc、go を使用した。





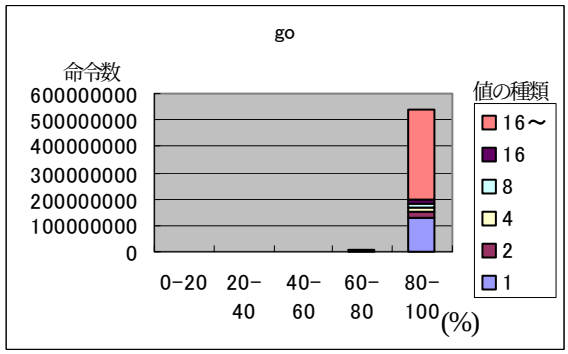


図 2 再利用可能性ととりうる値の個数

図 2 中の縦軸は動的命令数を、横軸は再利用可能

性を表している。ここで再利用可能性は、各静的命令についての

既出の入力による実行数/全実行回数×100(%)

の値である。また、とりうる入力値の個数について分類した。

ここでは全動的命令の入力値を保存しているため、すべてのベンチマークで再利用可能性の高い命令がほとんどを占めている。再利用可能性の高い命令でも、とりうる値の種類は多いものと少ないものがあることがわかった。

4. RB の改良

[1]において、エントリの置換アルゴリズムは FIFO であった。しかし、3.の結果から、実行時の実際の値局所性を加味したエントリ管理を行ったほうがよりよい再利用率が得られる可能性があると考えられる。また、図からわかるように各命令で複数の値が再利用されている。そこで、以下のように、1 つの命令に対し複数のエントリを保持し(図)、再利用率を考慮した、エントリ置換アルゴリズムを提案する。

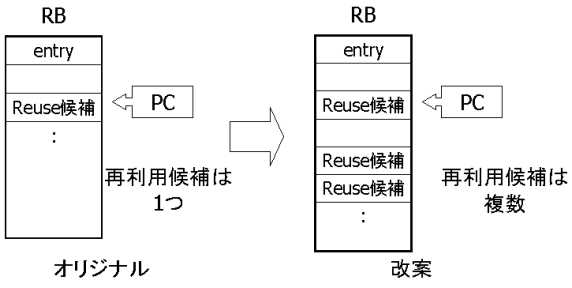


図 3 1 命令に対応するエントリの複数化

1 命令に対し複数のエントリを割り当てる場合、無制限にエントリの複数保持を許すと、場合によっては再利用性の低い命令のエントリが再利用性の高い命令のエントリを追い出してしまう可能性がある。そのため、上限(以下 Cap と呼ぶ)を設ける。

再利用度を置換に反映させるため再利用度カウンタ(Ref カウンタ)を RB の各エントリに追加する。登録時に 0 にし再利用されるごとに+1 する。

これに更に Age カウンタを追加する。これは再利用されたことがあるが古いエントリを掃き出すためのもので、新規登録時はフィールドの全ビットを ON、RB 参照ごとに-1 とし、これを優先度に加味することにより古いエントリがいつまでも居座るのを防ぐ。吐き出しエントリの選択には、上記 2 つのカウンタを使い優先度を算出。優先度の最低のものを掃き出す(図 4)。

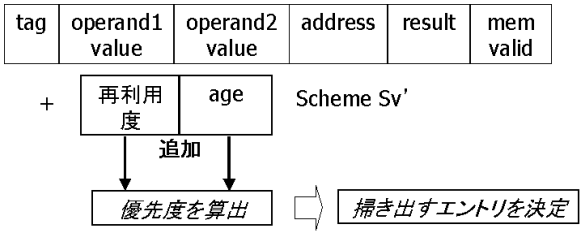


図 4 カウンタ付き RB エントリ

5. 実験

5.1 実験環境

表 1 シミュレータのパラメータ

Host	SimpleScaler3.0(sim-outorder)
Instruction Fetch	4 insts/cycle
I-cache	16KB, directmapped, 32byte/block, 6 cycle miss latency
Branch predictor	2048 BTB entries with 2bit satulating counter Speculative exection mechanism: out of order issue/commit of 4 op/cycle, 32 entry reorder buffer, 32 entry load/store queue. Maximum of 8 unresolved branches. Loads execute only after all the preceding store address are known. Valuesbypassed to loads from matching stores ahead load/store queue.
Archtected Registers	32 integer, hi, lo, 32floating point, fcc.
Function units	4-integer ALUs, 2-load/store units, 4-FP adders, 1-integer MULT/DIV, 1-FP MULT/DIV
Function unit latency (total/issue)	integer ALU 1/1, load/store 1/1, integer MULT 3/1, integer DIV 20/19, FP adder 2/1, FP MULT 4/1, FP DIV 12/12, FP SQRT 24/24.
D-cache	16K 2-way set associative, 32 bytes/bock, 6 cycle miss latency, Dual ported, non-blocking interface, one outstanding miss per register.

SimpleScalar toolset[12]のサイクルレベルシミュレータ(Sodani らが[1]で用いたものと同じ)に、改良した値再利用機構を組み込み、その効果を調べた。ベンチマークプログラムには準備実験で用いた SPEC95 の compress, gcc, go に加え、SPEC2000 の mcf と parser を用いる。性能の指標には IPC(Instruction Per Cycle)を用いた。シミュレータのパラメータはデフォルトである(表 1)。

表 2 ベンチマークプログラム

Program	Input	Insts	IPC *1
compress	< 25000 e 2231	80M	1.7144
gcc(cc1)	-O 1stmt.i	279M	0.9202
go	50 9 2stone9.in	548M	0.8791
mcf	test(inp.in)	201M	1.1169
parse	test(2.1.dict -batch <test.in)	200M *2	1.5780

*1 RB 無しの値

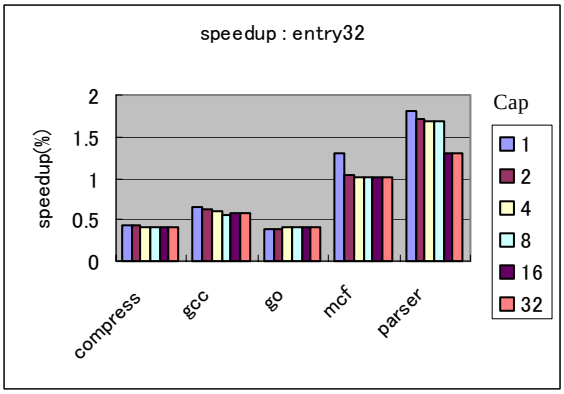
*2 最初の 240M をスキップ、440M で打ち切り

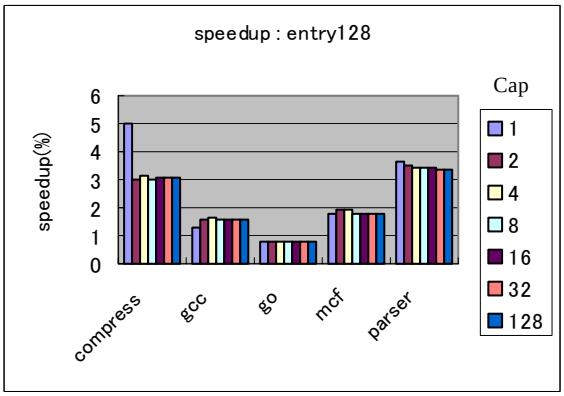
5.2 1 命令対応エントリの複数化

初めに 1 命令に対し複数のエントリを割り当てた場合の効果について調べた。図は、全エントリ数が 32, 138, 1024 それぞれの場合についての速度向上である。ここで置換アルゴリズムには FIFO を用いた。Cap は 1 命令に対応するエントリの上限を示している。speedup は RB なしの場合に対する IPC の向上率を示しており、

speedup=((IPC_RB 有) − (IPC_RB 無))/(IPC_RB 無)×100(%)

である。





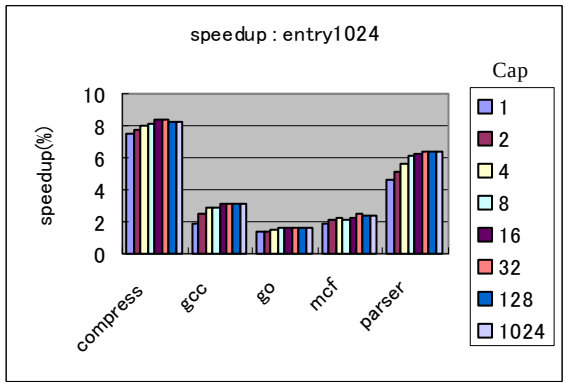
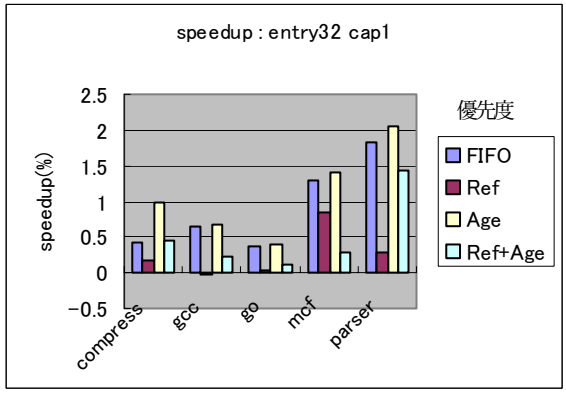


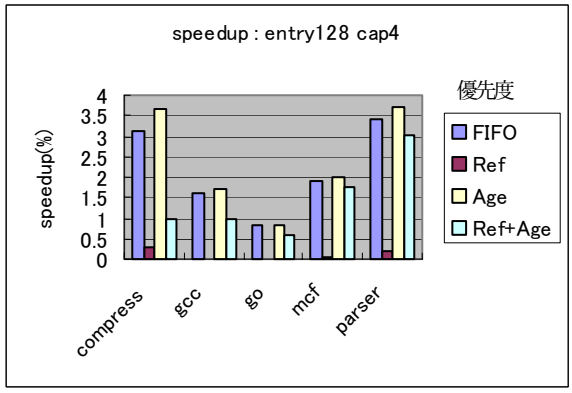
図 5 1 命令に複数エントリを割り当てた時の効果

全エントリ数が 32, 128 の場合は 1 命令対応エントリの複エントリ化の効果が見られなかったり、従来の単エントリ時よりも劣る結果となったが、1024 エントリの場合には単エントリの場合よりも+1%前後よい結果が得られていることがわかる。

5.3 優先度によるエントリ置換

次に、参照度(Ref)カウンタと Age カウンタの効果について調べた。今回、優先度算出には両カウンタの重みつき加算値を用いることにした。





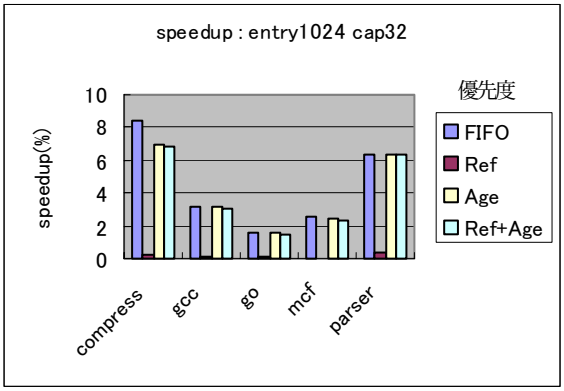


図 6 優先度による RB のエントリ置換の効果

まず、Ref カウンタのみ、Age カウンタのみ、Age+Ref(重みは均等)の 3 パターンの優先度を用いたエントリ置換を行った時の性能評価を行った。なお、Age カウンタのみの場合は LRU に相当する。ここで Cap は前の実験で最適と思われる値(32 エントリ:Cap=1、128 エントリ:Cap=4、1024 エントリ:Cap=32)を使用した。カウンタの長さはそれぞれ全エントリ数がおさまる幅(全エントリ数が 128 なら 7 ビット)とした。これは、全てのエントリに異なる優先度割り振るための幅ということである。

結果を速度向上の順にまとめると、

Age カウンタのみ ＞ FIFO ＞ Ref+Age ＞ Ref カウンタのみ ＞ RB なし

というような結果なった。この傾向はおおむね全エントリ数にかかわらずに見られる。ただし、全エントリ数が 1024 の場合はその差は小さく、compress のように FIFO の方が高い性能を示しているものもある。

実行時のエントリの登録/掃き出しの様子を見たところ、以下の 2 点が Ref カウンタのみの場合の速度向上が乏しい原因となっていた。

・早い段階である程度再利用されたエントリに新規エントリが勝てない

・Ref カウンタのみでは同じ優先度をもつエントリが複数存在してしまう

1 点目は提案の時点で予測していた通りである。成長する可能性のある若いエントリが成長する前に掃き出されてしまい、エントリ全体を有効に利用できないことが足かせになっている。

2 点目に関してだが、RB の全エントリが使われるよりも前にある程度の再利用が生じない限りは、このような状況が発生するのは当然といえる。これがなぜ問題にかというと、優先度が最も低いエントリが複数ある場合に、構造次第ではそれらのエントリから常に同じエントリを選択してしまい、実際に入れ替わるエントリが特定の 1 つになってしまう可能性があるためである。今回は構造の複雑化を考え、あえてこの点には対応しなかった。対処法として、そのグループ内で FIFO を適用するといった対処法が考えられるが、構造的に非常に複雑になることは避けられない。この問題は FIFO では発生しない。また、LRU と等価な Age のみの場合も発生しない。

2 点目に関しては Ref+Age の場合にも起こりうるが、すべてのエントリの Age が常に変化しつづけるため、Ref のみの場合のような状況は起こりにくい。

Ref+Age の場合が Age カウンタのみの場合に劣るのは上記の Ref カウンタが足を引っ張る形になっているためであった。

ここまでの結果から、Ref カウンタと Age カウンタを両方とも使用する場合、Ref カウンタのみの場合は FIFO のみよりも性能が悪く、Age カウンタのみの場合は FIFO のみよりも性能がよいことから、Age カウンタに大きな重みをつけたほうがよいことが分かる。そこで、次にカウンタの重み付けについて変化させてみた。ここでは、Cap のみで最も効果のあった 1024 エントリについて取り上げる。Cap は結果がほぼ頭打ちとなった Cap=32 とした。重みを考慮した優先度算出式は以下のものを用いた。

$$\text{優先度} = \text{Age} \times 2^{N} + \text{Ref}$$

たとえば N=2 のときは 4 倍することに相当する。倍はシフトで容易に実装できるためこの式を用いることにした。また、計算はオーバーフローが起こらないだけのビット幅(全エントリ数が 1024、N=2 のときは 10+2=12 ビット)で計算を行うものとする。Age と Ref の重なる部分は通常の加算を行う。

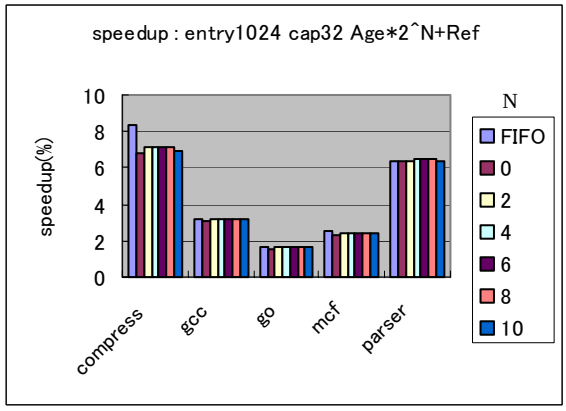


図 7 カウンタの重み付けの効果

図 7 は全エントリ数 1024、Cap=32 の場合の、速度向上が最大の時の値を示したものである。N はそのときの Age の重みである。N=6 付近で Age のみの場

合(N=10)よりも速度向上が見られる。また、compress、mcf 以外は、FIFO の場合を上回っている。

6. まとめ

本研究では値再利用機構の ReuseBuffer(RB)の改良を目指し、次の 2 点の手法を試みた。

(a) 1 つの命令に対応する RB のエントリを複数保持

(b) 再利用度、生存期間に基づいて算出した優先度によるエントリ置換

(a)は、エントリ数が少ない(32～128)場合には効果が無く、かえって悪くなることもあった。しかし、エントリ数が多い(1024)場合はオリジナルの FIFO のみの場合に比べて更に 1%前後の速度向上が見られた。また、上限(Cap)はエントリ数によって最適値が異なることが分かった。

(b)は、再利用度(Ref)のみでは性能がかえって悪くなり、生存期間(Age)のみでは FIFO よりもややよい結果が得られた。また、両者を組み合わせ適切な重みをつけて加算したものを優先度とすることにより、Age のみよりもわずかながらよりよい結果が得られた。

性能向上の度合いと機構の複雑さ考えると、(b)は、さほど有効な手法ではないと思われる。優先度の算出方法に関してはまだ研究の余地があるが、機構の複雑さを現状よりもおさえるのは困難であろう。

一方、(a)のほうは、前述のようにエントリ数の多い場合に今回取り上げた全てのベンチマークプログラムで効果が得られた。こちらに関して、今回は命令対応のエントリの上限値を固定値としたが、実際のプログラムごとの最適値は異なることが考えられる。これらの値の定性的な評価が必要となるだろう。また、特定命令に対応する現在のエントリ数を、その命令の局所性の指標にし、RB の有効利用に活用するといったことも考えられる。ただし、過度に機構が複雑化した場合、実装が困難になってしまう点は否めない。

これらの点が今後の課題である。

参考文献

[1] Avinash Sodani, Gurindra S. Sohi Dynamic Instruction Reuse Proceedings of the 24th International Symposiom on Computer Architecture(ISCA), June, 1997

[2] Mikko H.Lipasti, Christopher B. Wilkerson, and John Paul Shen Value Locality and Load Value Prediction In Proc. of ASPLOS VII, September, 1996

[3] 斎藤史子, 山名早人："投機的実行に関する最新技術動向", 情処研報(ARC), Vol.2001, No.116

[4] H.Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. Proc. of the 29th Annual International Symposium on Microarchitecture. Pp.227-237, December 1997.

[5] Y.Sazeides, J.E.Smith. Modeling Program Predictability. 25th ISCA pp.73-84. 1998.

[6] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction Using Hybrid Predictors. International Symposium on Microarchitecture 1997, pp.281-290.

[7] Jian Huang, David Lilja. Exploiting Basic Block Value Locality with Block Reuse. Proceedings of the 5th IEEE Int'l Symposium on High Performance Computer Architecture (HPCA-5), Orlando, FL, Jan, 1999.

[8] Daniel A. Connors, Wen-mei W. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results Proceedings of the 32nd International Symposium on Microarchitecture, November, 1999

[9] Jian Huang, David J. Lilja. Extending Value Reuse to Basic Blocks with Compiler Support. the IEEE Transactions on Computers, April, 2000.

[10] 中島康彦, 緒方勝也, 正西申悟, 五島正裕, 森眞一郎, 北村俊明, 富田眞治: "関数値再利用および並列事前実行による高速化技術", 情報処理学会論文誌:ハイパフォーマンスコンピューティングシステム, HPS5, pp.1-12, Sep. (2002).

[11] Youfeng Wu, Dong-Yuan Chen Jesse Fang. Better Exploration Region-Level Value Locality with Integrated Computation Reuse and Prediction. International Conference on Computer Architecture the 28th annual international symposium on Computer Architecture. 2001

[12] SimpleScalar LLC http://www.simplescalar.com/

[13] 佐藤寿倫, 有田五次郎. タグビット幅を考慮したデータ値予測機構のハードウェア量削減. 信学技報 CPSY 2000-3