計算機アーキテクチャ　109－1
（１９９４．１２．１３）

ロード命令の先行実行とその評価

小沢年弘[†]　　西崎慎一郎[††]　　木村康則[†]

[†](株) 富士通研究所　　[††](株) 富士通ソーシアルサイエンスラボラトリ

メモリ系のオーバーヘッドを低減するために、我々は、非数値計算プログラムに適した、ロード命令の先行実行を行なうアルゴリズムを開発した。このアルゴリズムでは、コンパイル時に、キャッシュ・ミスを起こしそうなロード命令をヒューリスティックスを用いて選び出す。そして、それらをなるべく早くスケジューリングするとともに、ロードしたデータを使用する命令との間に、なるべく多くの命令をスケジューリングする。このアルゴリズムを試作し、評価した結果、多くのプログラムで、速度向上が認められ、２０％近く速度向上するものもあった。さらに、プロセッサが同時に発行できる命令数とロード命令の先行実行の効果についても測定した。

Preload Technique and its Evaluation

ToshihiroOzawa[†], *Shin'ichiroNishizaki*[††], *YasunoriKimura*[†]

[†]FUJITSU Laboratories LTD. [††]Fujitsu Social Science Laboratory Ltd.

1015. Kamikodanaka Nakahara-ku, Kawasaki 211, Japan

email: ozawa@flab.fujitsu.co.jp

We have developed an efficient preload algorithm for non-numerical programs to hide memory latency. It uses heuristics that what kind of load instructions tend to cause a cache miss on non-numerical programs. If a load instruction is selected by the heuristics, it is scheduled early and other insturctions are tried to move into between the load instruction and the instructions which use the data loaded. We have implemented the algorithm and evaluated it. Most of programs we have tested speed up. The overall speedup is up to nearly 20 %. Also, we metion the relationship between the effect of preload and the number of instructions issued at the same time.

1 はじめに

近年、プロセッサ設計技術の高度化、半導体製造技術の進歩によって、プロセッサ性能と、メモリ系性能のバランスが変化している。一般に、プロセッサの速度向上にくらべ、メモリ系の速度向上が低く、実行時間に占めるメモリ系のオーバーヘッドをいかに低減するかが重要になっている。

メモリ系オーバーヘッドを低減する方法としては、キャッシュ・メモリ・サイズを大きくするばかりでなく、配列へのアクセスを小さい範囲ごとに分けて行なう blocking と呼ばれる方法 [1] や、データをメモリからキャッシュまで持ってくる prefetch 命令を用い、ロード命令発行時にはデータがキャッシュに載っているように、ロード命令より十分前に prefetch 命令を挿入する方法 [2] などが提案されている。

しかし、blocking では、配列へのアクセス・パターンが静的に判定できる必要があり、数値計算などの制御構造が比較的単純なプログラムには適用できるが、非数値計算プログラムなどの制御構造がより複雑なプログラムには適用が困難である。また、prefetch 命令を用いる方法では、prefetch 命令それ自身と、データ・アドレスを計算する命令を挿入しなければならず、実行命令数の増加が逆に性能を落す可能性がある。特に、非数値計算プログラムでは、分岐命令の頻度が高く、prefetch 命令が投機的命令[1]になってしまうので、この可能性が高くなる。

メモリ系オーバーヘッドを低減するもう一つの方法として、ロード命令の先行実行（preload）がある。この方法では、コンパイラによって、ロード命令とロードしたデータを使用する命令（使用命令と呼ぶ）の距離を離したコードを生成し、non-blocking ロード命令[2]を備えたハードウエアで実行することにより、キャッシュ・ミス時に他の命令を並列に実行し、メモリ系オーバーヘッドを隠蔽する方法である。この方法は、prefetch の方法と似ているが、prefetch 命令やデータ・アドレスを計算する命令を新たに挿入しないので、実行命令数の増加を抑えることが期待できる。

我々は、ロード命令の先行実行を効率的に行なわせるコンパイラを GNU の SPARC 用 C コンパイラをベースにして試作した。本稿では、実装したロード命令の先行実行の方式と、整数系ベンチマーク・プログラムである、Specint92 使った定量的な評価を述べる。

2 ロード命令の先行実行

ロード命令の先行実行により、メモリ系のオーバーヘッドを効果的に隠すためには、以下の項目が重要である。

1. 先行実行すべきロード命令の選択

   ロード命令がキャッシュ・ミスを起こさないならば、メモリ系のオーバーヘッドは生じない。逆に、それらを早期に発行することは、レジスタのライフタイムを不必要に伸ばし、レジスタ・アロケーションの自由度を下げたり、他の最適化が行なえなくなるなどの悪影響を与える。さらに、そのようなロード命令を投機的に移動するならば、メモリ系のオーバーヘッドは減らないのに、実行命令数だけが増えてしまうことになりかねない。このように、キャッシュ・ミスを起こしそうもないロード命令までを早期に発行することは、性能低下を引き起こす。従って、キャッシュ・ミスを起こしそうなロード命令を予測、選択することが重要である。

2. メモリ系オーバーヘッドを低減させる命令スケジューリング

   キャッシュ・ミスを起こしそうなロード命令と使用命令との距離をなるべく離してスケジューリングすることにより、メモリ系のオーバーヘッドを隠すことがで

[1]ある条件が成立した時に実行される命令を、その条件がなり立たないときにも実行される位置に置いたとき、その命令を投機的と呼ぶ。一般に、分岐命令の下から上に命令を移動することは、投機的な命令移動になる。

[2]ロード命令がキャッシュ・ミスを起こした時、その完了を待たずに後続命令の実行が行なえるロード命令。ただし、後続命令が、ロード命令がロードしてくる値を使用する場合には、その命令の実行は、キャッシュ処理の完了まで待たされる。

きる。プログラムの意味を変えず、実行命令数も不当に増加させない範囲で、如何にこれらの命令の距離を離してスケジューリングするかが問題となる。

2.1 先行実行すべきロード命令の選択

2.1.1 キャッシュ・ミス予測のヒューリスティックス

数値計算プログラムでは、メモリ参照は、配列への規則的な参照により行なわれることが多く、キャッシュ・ミスを静的に求めることができる場合もある。しかし、非数値計算プログラムなど、一般のプログラムでは、メモリ参照の規則性は少なく、ミスの完全な予測を静的にすることは、困難である。そこで、ある程度の確率で、キャッシュ・ミスが高い場合を選り分けるヒューリスティックスを得ることを目的とした。

我々は、キャッシュ・シミュレータ [3] により、Specint92 ベンチマーク・プログラムにおいて、キャッシュ・ミスを起こす命令とその回数を測定した。その結果、ロード・アドレスがどのように生成されてきたのかということに注目することで、次の2つの場合にキャッシュ・ミスを起こすことが多いというヒューリスティックスを得た。

- **list access:** ロードしたデータをベース・アドレスとして、さらにロードを行なう場合
- **big stride access:** ループ内で、小さくない stride でデータを読み出す場合

それぞれの例を上げる。

list access:

```
ld [ R0 + offset1 ], R0      (11)
add R1,-1,R1                 (12)
cmp R1, 0                    (13)
bl  LABEL                    (14)
ld [ R0 + offset2 ], R0      (15)
```

この例では、命令（１１）で読み出したデータ（レジスタ R0 に入っている）を、命令（１５）で、ベース・アドレスとして使用している。この時、命令（１５）では、キャッシュ・ミスが起こり易い。

list access が現れるのは、各データをポインタで結んで、その探索、更新などの処理を行なう場合が考えられる。この場合には、各データの空間的なローカリティが低いと予想される。

big stride access:

```
LABEL:
      call func(R1)          (21)
      add R2, R0, R2         (22)
      ld [R2], R3            (23)
        :
      add R1,-1,R1           (24)
      cmp R1, 0              (25)
      bplus  LABEL           (26)
```

この例では、レジスタ R1 の値が 0 になるまで、（２１）～（２６）の命令を繰り返し実行するループ構造になっている。命令（２１）では、レジスタ R1 の値を引数として、関数 func を呼び出す。関数 func では、その結果をレジスタ R0 に格納するものとする。命令（２３）では、ループが一周する毎に、前にアクセスしたアドレスをベースとして、それからレジスタ R0 の値だけずれたアドレスをアクセスする。このずれ（オフセット）が大きな値であったり、可変である場合、命令（２３）でキャッシュ・ミスが起こり易い。

big stride access（以下、**BSA** と略す。）が現れるのは、配列の各要素を順に参照し、かつ一つの要素の大きさが、（キャッシュ・ブロックに比べて）比較的大きい場合や、ハッシュ・テーブルをハッシュ関数により参照する場合などが考えられる。これら場合には、各参照の空間的なローカリティが低いと予想される。

2.1.2 キャッシュ・ミス予測のヒューリスティックスの評価

全ロード命令の内、**list access**、**BSA** に分類されるロード命令の静的割合が少なく、かつキャッシュ・ミスの大部分が、それらによっ

表 1: ロード命令の種類別の静的な割合（%）

program	list access	BSA	合計
008.espresso	16.3	8.6	24.9
022.li	19.8	10.3	30.1
023.eqntott	20.4	7.0	27.4
026.compress	7.0	5.9	12.9
072.sc	13.1	19.8	32.9
085.gcc	28.3	8.4	36.7

表 2: ベンチマーク・プログラムへの入力

program	入力
008.espresso	input.ref/bca.in
022.li	input.veryshort/li-input.lsp.8
023.eqntott	input.ref/int_pri_3.eqn
026.compress	input.ref/in
072.sc	input.ref/loada1
085.gcc	input.ref/1cexp.i

て引き起こされるならば、前節で上げたヒューリスティックスは有効であると考えられる。

コンパイル時に制御の流れを追い、ロード・アドレスを解析することによって、各ロード命令を、**list access**、**BSA**、その他に分類した。ただし、解析の複雑さから、natural loop[3]以外のループが出てきた時には、解析を中止しその他に分類している。また、natural loop に含まれるロード命令で、そのロード・アドレスを特定できない場合には、**BSA** に分類している。

表1に、全ロード命令のうち、**list access**、**BSA** に分類されたロード命令の命令数[4]の静的割合を、各ベンチマーク・プログラムごとに示す。プログラムによる割合の差はあるものの、この2つの種類に属するのは、合わせて全ロード命令の約 30 %程度であることが分かる。

[3]入口が一つだけのループ。C, Fortran などの言語で記述されたプログラムにおいては、ほとんどのループが natural loop となる。

[4]厳密には、最終的な機械語のロード命令の数ではなく、コンパイラのフロー解析時の中間表現におけるロード命令の数である。コンパイルの処理が進むと、この数は多少増減する。

表 3: ロード命令の種類別キャッシュ・ミス回数の比（%）(cache size 32Kbytes, direct map)

program	list access	BSA	合計
008.espresso	10.2	78.1	88.3
022.li	15.3	50.0	65.3
023.eqntott	90.6	0.3	90.9
026.compress	1.4	97.7	99.1
072.sc	64.3	15.6	80.1
085.gcc	29.8	25.4	55.2

表 4: ロード命令の種類別キャッシュ・ミス回数の比（%）(cache size 32Kbytes, 2way)

program	list access	BSA	合計
008.espresso	9.6	85.8	95.4
022.li	9.3	69.6	78.9
023.eqntott	92.1	0.2	92.3
026.compress	1.2	98.7	99.9
072.sc	65.6	15.4	81.0
085.gcc	28.8	32.5	61.3

また、Specint92 の入力データの内、表 2 に示したものを入力として、各ベンチマーク・プログラムを実行し、ロード命令によるキャッシュ・ミス回数を、ロード命令の種類別に集計した。

データ・キャッシュの構成が、

- Cach size: 32Kbytes, block size: 32bytes, direct map cache
- Cach size: 32Kbytes, block size: 32bytes, 2way set associative cache

の場合についての結果を、それぞれ表 3、表 4 に示す。ただし、命令キャッシュは、ミスをしないものとした。

大部分のプログラムで、どちらのキャッシュ構成においてもロード命令によるキャッシュ・

ミスの８０％以上が、**list access**、**BSA** に属するロード命令で起きていることが分かる。085.gcc で全般的に値が低くなっているのは、natural loop 以外のループの中にキャッシュ・ミス回数の多いロード命令があるが、今回の解析では解析を簡単にするために、それらをその他分類してしまっているためである。

以上より、前節で述べたヒューリスティックスに属する、全体の約３０％程度のロード命令に対して、キャッシュ・ミスに関するなんらかの処置を行なえば、全キャッシュ・ミスの約８０％以上に対して処置を行なったことになる。従って、先行実行の対象とするロード命令をヒューリスティックスに属するロード命令だけに絞ることにより、他の最適化を妨げたり、実行命令をむやみに増加したりしないで、キャッシュ・ミスのオーバーヘッドを効果的に減少させることができると考えられる。

2.2 メモリ系オーバーヘッドを低減させる命令スケジューリング

キャッシュ・ミスを起こすと予測されたロード命令と、その使用命令との距離をなるべく離し、メモリ系オーバーヘッドを低減させるために、ベーシック・ブロック内の命令スケジューリングと、ベーシック・ブロック間に渡る命令移動を行う。

2.3 ベーシック・ブロック内スケジューリング

ベーシック・ブロック内スケジューリングでは、キャッシュ・ミスを起こすと予測されたロード命令は、早期に発行されるようにスケジュールされ、使用命令は、なるべく遅く発行されるようにスケジュールされる。

2.4 ベーシック・ブロック間命令移動

キャッシュ・ミスを起こすと予測されたロード命令とその使用命令との距離が十分離れていないならば、ベーシック・ブロック間に渡って、以下のような命令移動を行なう。

1. **キャッシュ・ミスを起こすと予測されたロード命令を先行ベーシック・ブロックに移動する。**ただし、実行命令数の増加を防ぐため、投機的移動は行なわない。つまり、ロード命令が元々あったベーシック・ブロック B_{form} が実行される時は、移動先のベーシック・ブロック B_{to} は、必ず実行され、かつベーシック・ブロック B_{to} が実行される時は、ロード命令が元々あったベーシック・ブロック B_{form} も必ず実行される関係になければならない。(図 1(1)move)

2. **キャッシュ・ミスを起こすと予測されたロード命令に対する使用命令を後続ベーシック・ブロックに移動する。**ただし、実行命令数の増加を防ぐため、投機的移動は行なわない。(図 1(2)move)

3. **キャッシュ・ミスを起こすと予測されたロード命令とその使用命令の間に、先行、または後続ベーシック・ブロックから命令を移動する。**ただし、実行命令数の増加を防ぐため、投機的移動は行なわない。(図 1(3)move)

4. **ループ内においてキャッシュ・ミスを起こすと予測されたロード命令とその使用命令の間に、後続ベーシック・ブロックから命令を投機的に移動する。**ループ内では投機的に移動するが、実行命令数の増加を押えるため、ループから抜ける可能性のある分岐命令（loop exit jump）を２つ以上越えては移動しない。また、より内部のループへの移動も行なわない。(図 2(4)move)

5. **ループ内においてキャッシュ・ミスを起こすと予測されたロード命令を先行ベーシック・ブロックに投機的に移動する。**ループ内では投機的に移動するが、実行命令数の増加を押えるため、ループから抜ける可能性のある分岐命令（loop exit jump）を２つ以上越えては移動しない。また、より内部のループへの移動も行なわない。(図 2(5)move)

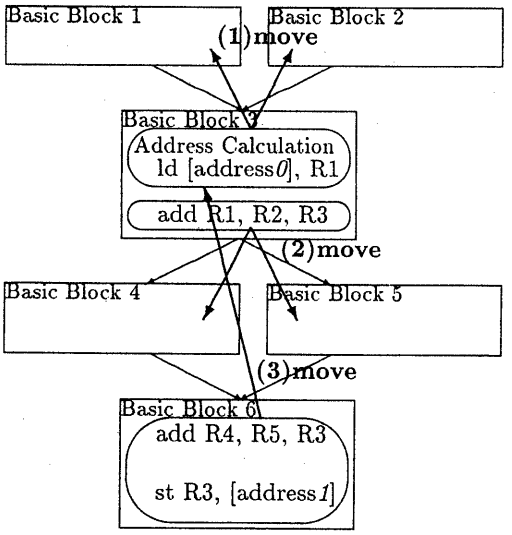

図 1: ベーシック・ブロック間命令移動の例

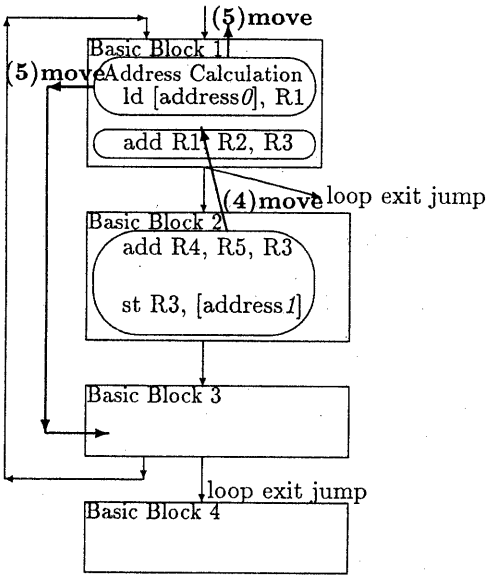

図 2: ループ内のベーシック・ブロック間命令移動の例

この移動を行なうと、不当なロード・アドレスでロード命令が発行される場合が出てくる。このような場合、通常のロード命令では割り込みを起こしてしまうので、この移動を行なう場合には、割り込みを発生させないロード命令を用意する必要がある。

これらの移動を行なう時、命令間の依存関係のために、ある命令を移動しないとロード命令、または使用命令の移動ができないならば、その命令も同時に移動する。例えば、図1において、Basic Block3 の ld 命令は、ロード・アドレスを計算する命令群（Address Calculation ）とともに先行ベーシック・ブロック（Basic Block1, Basic Block2）に移動させられる。また、Basic Block6 の add 命令、st 命令は、それぞれ単独ではレジスタ R3 の依存関係を壊してしまうので ld 命令とその使用命令の間に移動できない。しかし、二つ一緒ならばレジスタ R3 のライフタイムが完全に使用命令の前に移るので、移動することができる。

3 ロード命令先行実行の評価

前章のようなスケジューリングを行なうスケジューラを GNU の SPARC 用 C コンパイラに組み込んだ。このコンパイラを preload コンパイラと呼ぶ。

オリジナルの GNU C コンパイラを使用した場合、Specint92 ベンチマーク・プログラムの実行時間が、キャッシュ・ミス時のレイテンシィの違いによりどのように変化するかを表5に示す。ただし、データ・キャッシュ構成は、大きさ 32Kbytes, direct map, ブロック・サイズ 32bytes であり、blocking ロード命令で、1 fetch, 1 issue の in order 実行のハードウエアを想定している。また、命令キャッシュは、ミスをしないものとした。

レイテンシィ０、つまり、キャッシュ・ミスなしの場合に比べて、026.compress では、レイテンシィ４０サイクルの場合、100%以上の実行時間の増加がある。これがメモリ系のオーバーヘッドである。

この blocking ロード命令での実行時間を１００とした時の、・ロード命令を non-blocking ロード命令にした場合のオリジナルの GNU C

表 5: キャッシュ・ミス時のレイテンシィによる実行時間の変化（%）(cache size 32Kbytes, direct map)

program	latency(cycles)				
	0	10	20	30	40
008.espresso	100	108	115	123	131
022.li	100	102	104	106	108
023.eqntott	100	105	109	114	118
026.compress	100	126	152	179	205
072.sc	100	114	128	141	155
085.gcc	100	104	109	113	117

表 6: non-blocking ロード命令による実行時間の変化（%）(cache size 32Kbytes, direct map)

program	latency(cycles)			
	10	20	30	40
008.espresso	98	98	97	97
022.li	99	98	97	97
023.eqntott	99	99	99	99
026.compress	94	91	91	91
072.sc	98	98	97	97
085.gcc	98	97	96	96

コンパイラによるコードの実行時間
を表6に示し、non-blocking ロード命令の効果を示す。また、
・ロード命令を non-blocking ロード命令にした場合の preload コンパイラによるコードの実行時間
を表7に示し、preload コンパイラの効果を示す。

一般に、non-blocking ロード命令を備えるだけで、1、2%から10%の性能向上がみられるが、preload コンパイラを使用することにより、さらに数%から10%程度の速度向上が得られている。特に、026.compress や 072.sc のようにメモリ系のオーバーヘッドが元々大

表 7: non-blocking ロード命令と preload コンパイラによる実行時間の変化（%）(cache size 32Kbytes, direct map)

program	latency(cycles)			
	10	20	30	40
008.espresso	95	95	95	95
022.li	100	99	99	99
023.eqntott	92	92	92	93
026.compress	82	81	83	84
072.sc	92	91	91	91
085.gcc	98	97	96	95

きいプログラムに対して大きな速度向上が得られている。メモリ系のオーバーヘッドが元々小さいプログラムに対しても、022.li を除いては、速度向上が得られている。広い範囲のプログラムで速度が向上したのは、投機的移動を制限して、実行命令数の増加を抑えているためである。

4 CPU 構成とロード命令先行実行

ロード命令先行実行の効果と CPU 構成、特にスーパスカラや、Branch Target Buffer(BTB) との関係について考察するために、CPU 構成を変化させて、測定・評価を行なった。

図3に、同時に発行できる命令数 (issue 数) を 1、2、4 と増やした時の GNU C コンパイラに対する preload コンパイラの速度向上率を示す。また、図4には、さらに BTB を加えた場合の速度向上率を示す。ただし、実行は、out-of-order で、BTB は、4Kbytes の大きさで 2bit 予測を行なうものとし、cache size 32Kbytes、レイテンシィ20サイクル、direct map のキャッシュのキャッシュ構成で測定した。また、issue 数が2以上では、ALU を2個にしている。

BTB がない場合、issue 数が上がると、preload の効果は増加する傾向にある。これは、preload のために投機的に移動した命令を他の命令と並

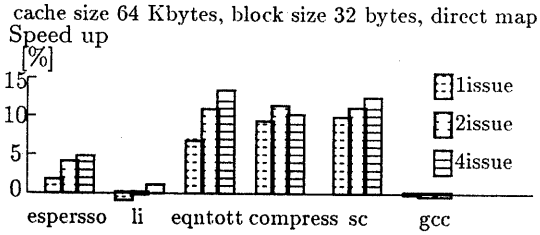

図 3: preload コンパイラによる速度向上の CPU 構成に対する変化（BTB なし）

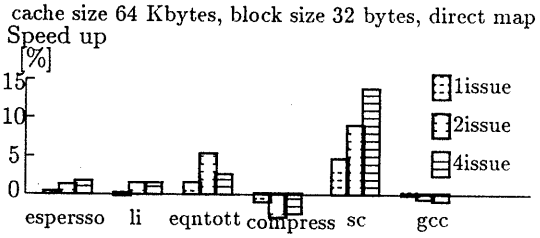

図 4: preload コンパイラによる速度向上の CPU 構成に対する変化（BTB あり）

列に実行できるようになり、投機的移動のオーバーヘッドを隠蔽できるためである。BTB があると、preload の効果が相殺されてしまう場合がある。これは、ループの繰り返しを越えてロード命令を移動にすることによって、preload の効果が得られることが多いためである。つまり、BTB を付けることによりループの繰り返しのための分岐を越えて、命令を（動的に）投機的に実行するようになる。これは、ちょうど、スケジューラがこの分岐を越えて、命令を（静的に）投機的に移動するのと同じ効果を持つことになる。このため、スケジューラの命令移動における、第４、５項目の移動の効果が、BTB によっても得られているためである。

5 まとめ

キャッシュ・ミスを静的に予測するヒューリスティックスとして、

- **list access:** ロードしたデータをベース・アドレスとして、さらにロードを行なう場合
- **big stride access:** ループ内で、小さくない stride でデータを読み出す場合

を見い出し、静的な割合で、ロード命令の約３０％が、このヒューリスティックスに当てはまること、また、大部分のプログラムにおいて、ロード命令によるキャッシュ・ミスの約８０％以上が、ヒューリスティックスに当てはまるロード命令で起きていることを見い出した。

このヒューリスティックスを使い、ロード命令の先行実行を行なうコンパイラ（preload コンパイラ）を試作した。

一般的な傾向として、

- preload で削除できるメモリ系オーバーヘッドは、メモリ系オーバーヘッドの４０％くらいまでで、全実行時間の２０％程度までであった。
- preload による効果は、CPU の並列度が上がると大きくなる傾向にある。ただし、out-of-order 実行、BTB による効果と相殺する場合が見られる。

6 今後の課題

非数値計算プログラムに対する評価や、prefetch 方式との比較などを検討していく予定である。

参考文献

[1] T. C. Mewry, and M. S. Lam. Design and Evaluation of a Compiler Algorithm for Prefectching. In Proceedings of 5th ASPLOS, 1991.

[2] W. Y. Chen, S. A. Mahlke, P.P. Chang, and W.W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In Proceedings of Microcomputing 24, 1991.

[3] 志村 浩也 他. スーパスカラプロセッサの性能評価 - Paratool -. 情報処理学会研究会報告 93-ARC-102-1, 1993.