

2015年1月27日 公聴会

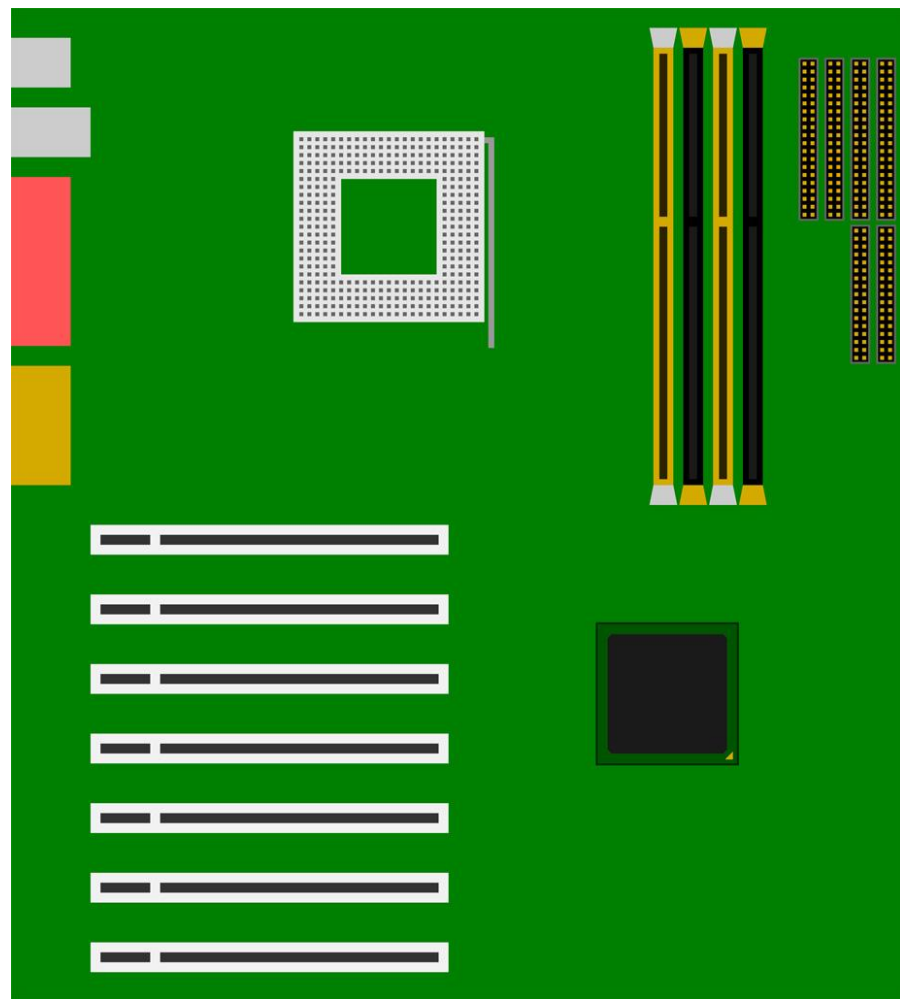


# Optimizing Utilization of Memory Hierarchy Based on Code Motion

---

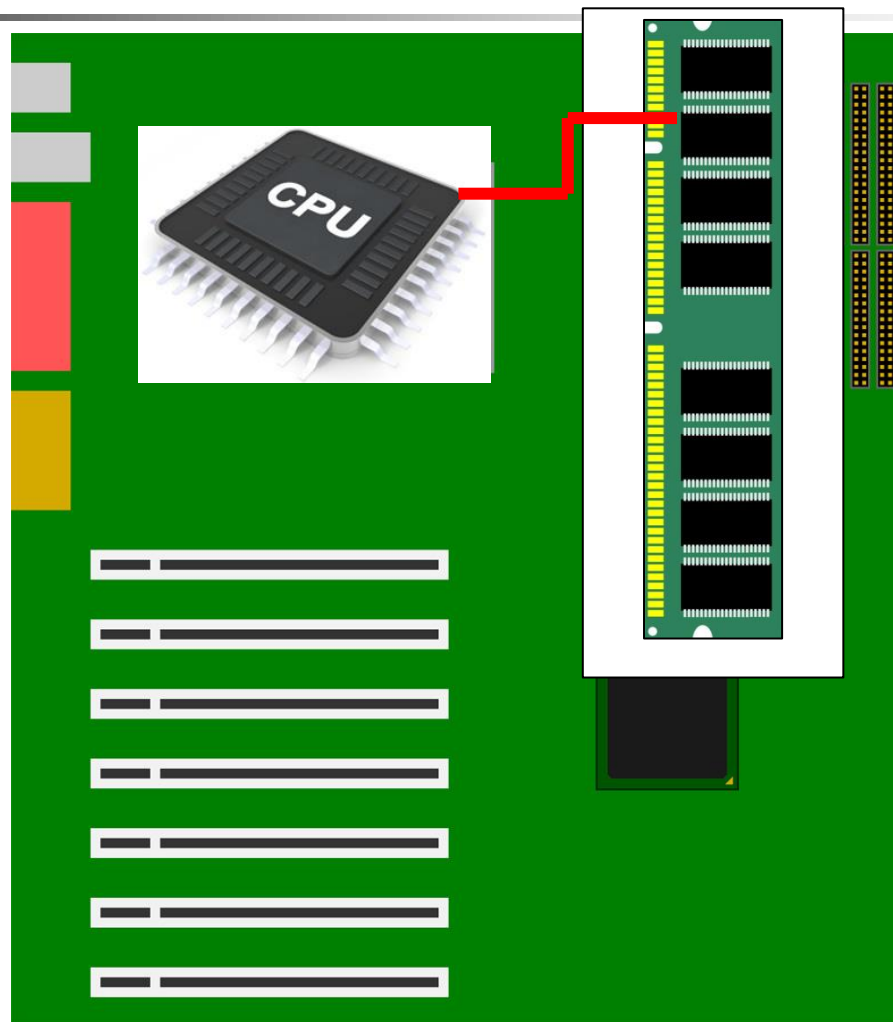
東京理科大学大学院 理工学研究科 情報科学専攻  
滝本研究室 博士後期課程 3年  
澄川 靖信

# コンピュータのメモリ構成

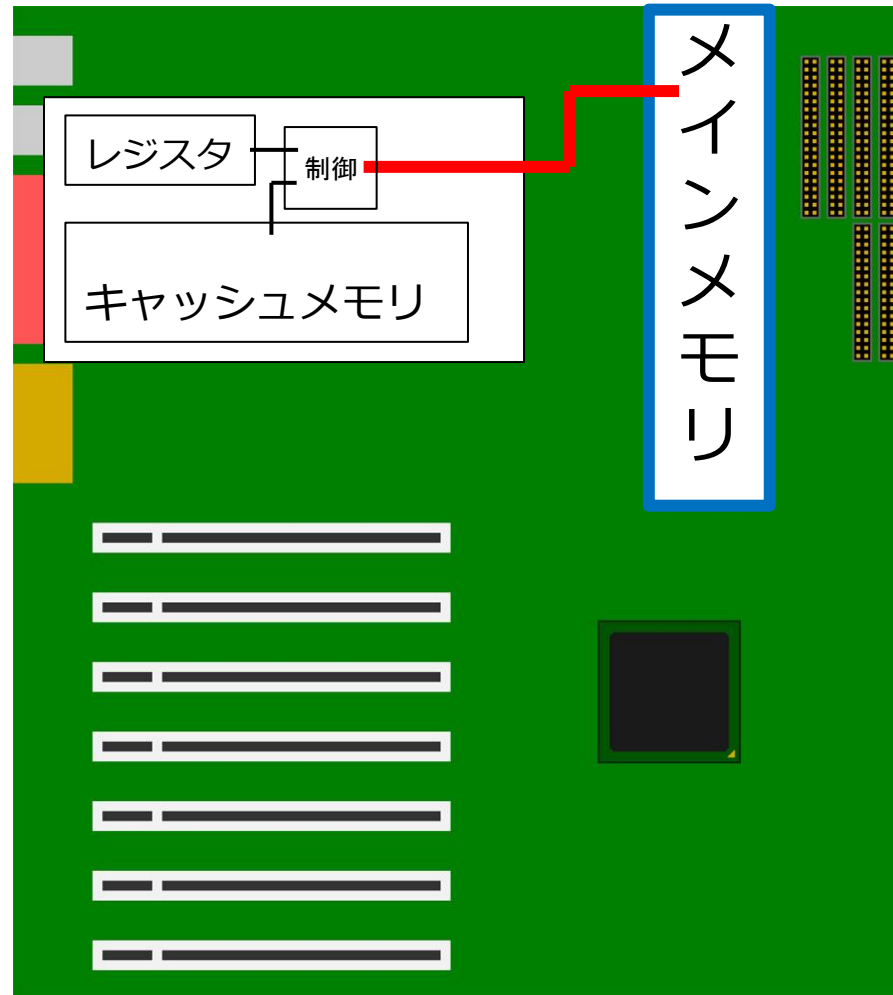


マザーボード

# コンピュータのメモリ構成



# コンピュータのメモリ構成





## データ転送の様子

---

```
main () {  
    x=a[i];  
}
```

Cプログラム

配列 $a[i]$ のデータ：メモリに配置  
変数 $x$ のデータ：レジスタに配置

※ 本発表で使用する全てのプログラムは  
C言語で書かれていると仮定する。

# データ転送の様子

```
main () {  
    x=a[i];  
}
```

制御

レジスタ

キャッシュ

メインメモリ

# データ転送の様子

```
main () {  
    x=a[i];  
}
```

a[i]のデータ  
はある？

制御

レジスタ

キャッシュ

メインメモリ

# データ転送の様子

```
main () {  
    x=a[i];  
}
```

キャッシュミス

制御

レジスタ

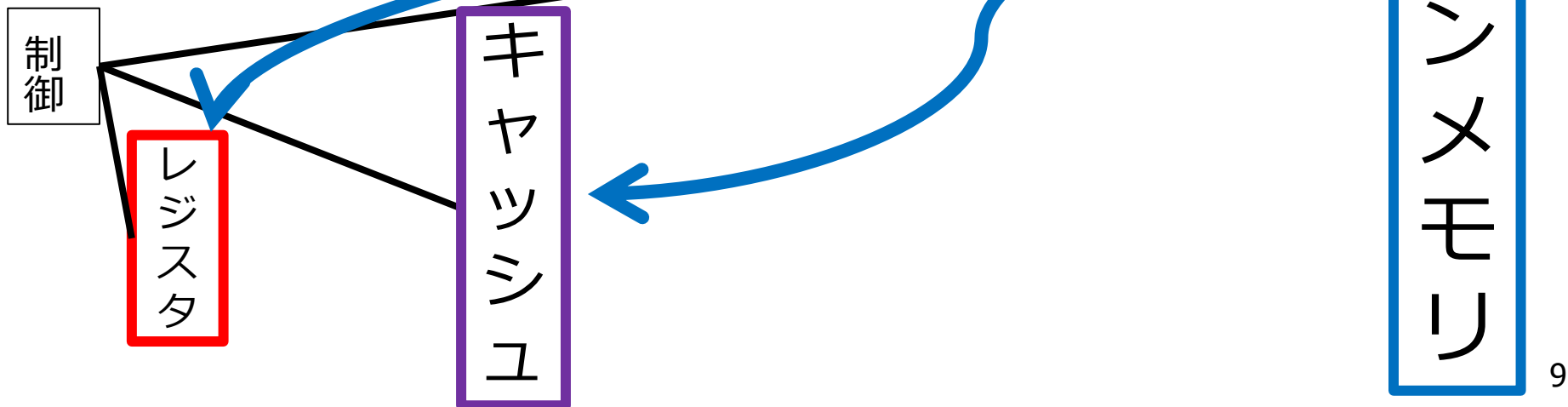
キャッシュ

メインメモリ



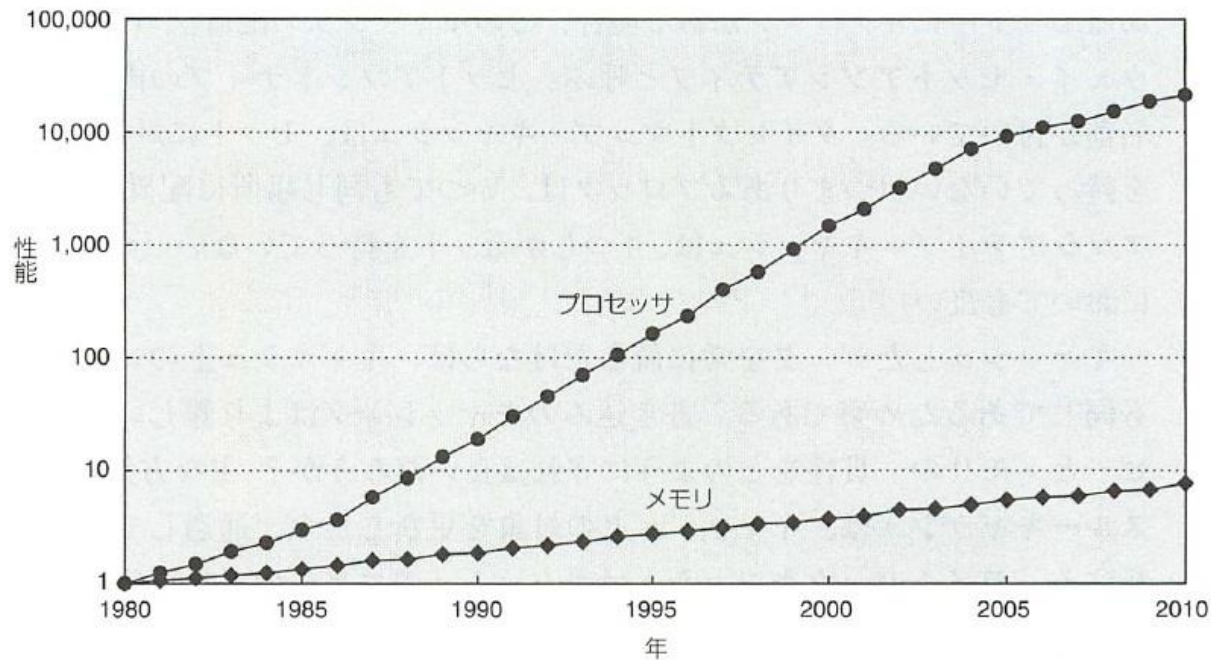
# データ転送の様子

```
main () {  
    x=a[i];  
}
```



# 問題点

- プロセッサとメインメモリの性能差が拡大.
  - メインメモリのアクセスによるペナルティ増加を意味する



1980年の性能を基準とした場合の、メモリとプロセッサの性能ギャップ



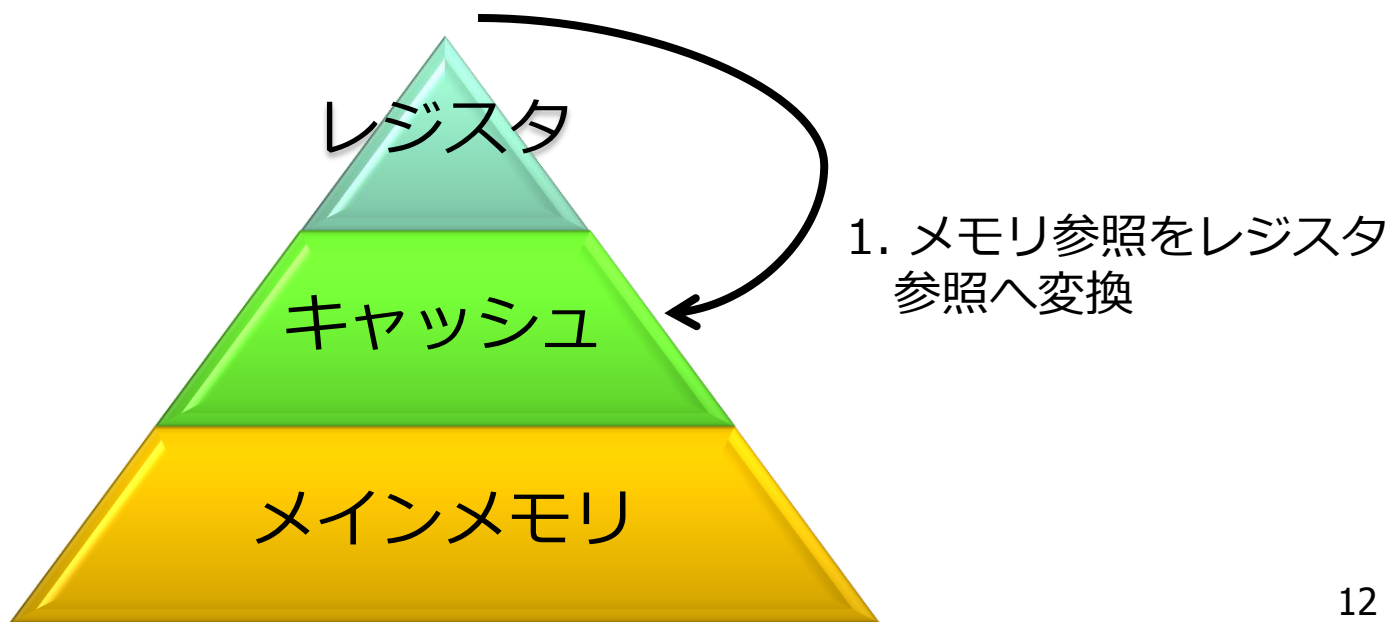
## 本研究の目的

---

- **メモリ階層を効率良く使用**するプログラムを生成するコンパイラのコード最適化の手法を提案する。
  1. メモリ参照をレジスタ参照へ変換
    - 不要に実行される配列参照を除去する
  2. キャッシュメモリのミス数の低減
    - 参照の局所性を利用

# 本研究の目的

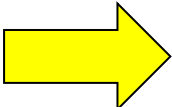
- **メモリ階層を効率良く使用する**プログラムを生成するコンパイラのコード最適化の手法を提案する。
  1. メモリ参照をレジスタ参照へ変換
    - 不要に実行される配列参照を除去する



# 本研究の目的

- **メモリ階層を効率良く使用**するプログラムを生成するコンパイラのコード最適化の手法を提案する。
  1. メモリ参照をレジスタ参照へ変換
    - 不要に実行される配列参照を除去する

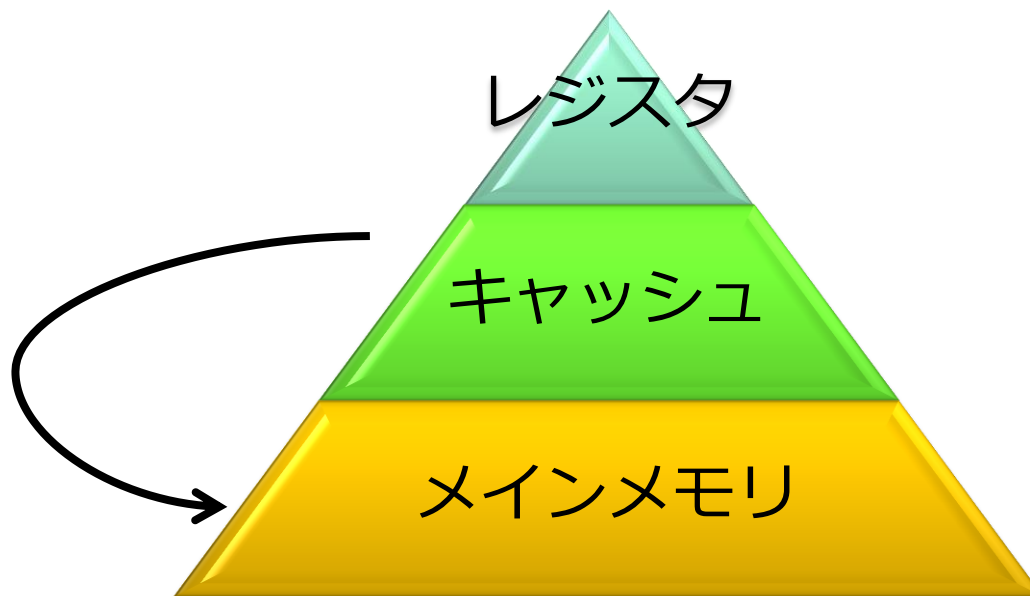
```
main() {  
    x=a[i];  
    y=a[i];  
}
```

  
a[i]を除去

```
main() {  
    x=a[i];  
    y=x;  
}
```

# 本研究の目的

- **メモリ階層を効率良く使用**するプログラムを生成するコンパイラのコード最適化の手法を提案する.
2. キャッシュメモリのミス数の低減
    - 参照の局所性を利用

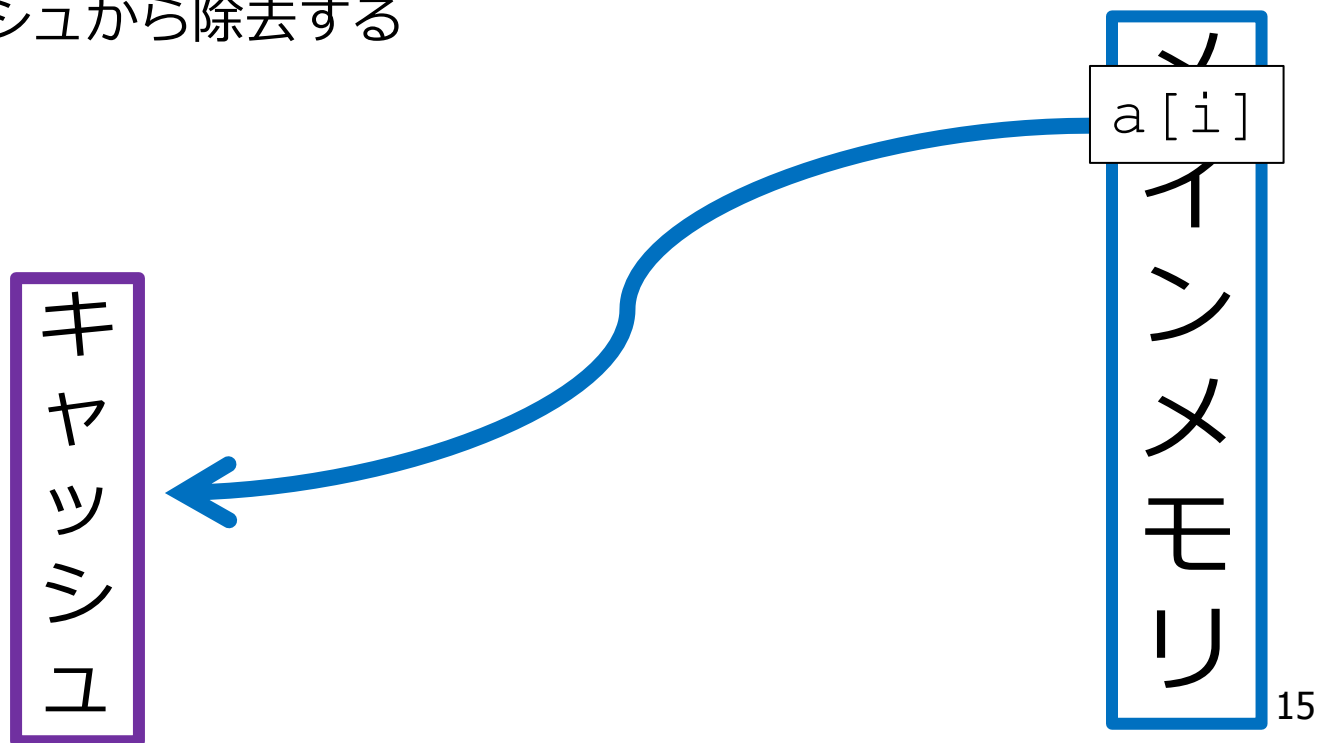


# キャッシュメモリのミス数の低減

- メモリ容量

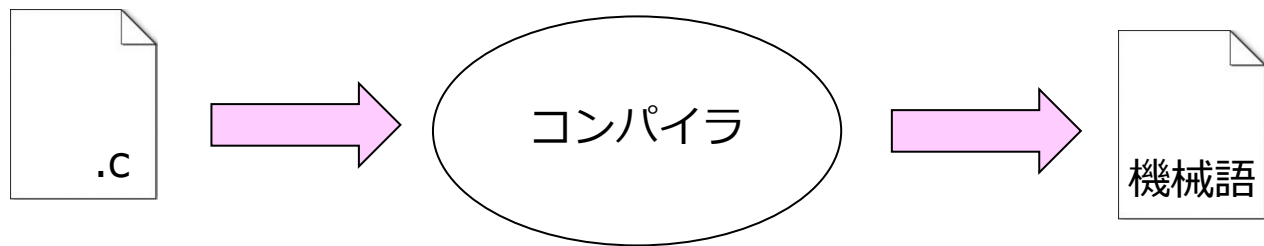
- キャッシュメモリ < メインメモリ

→ キャッシュメモリにデータを配置するとき、古いデータをキャッシュから除去する



# 実現方法

- コンパイラのコード最適化の手法として実現

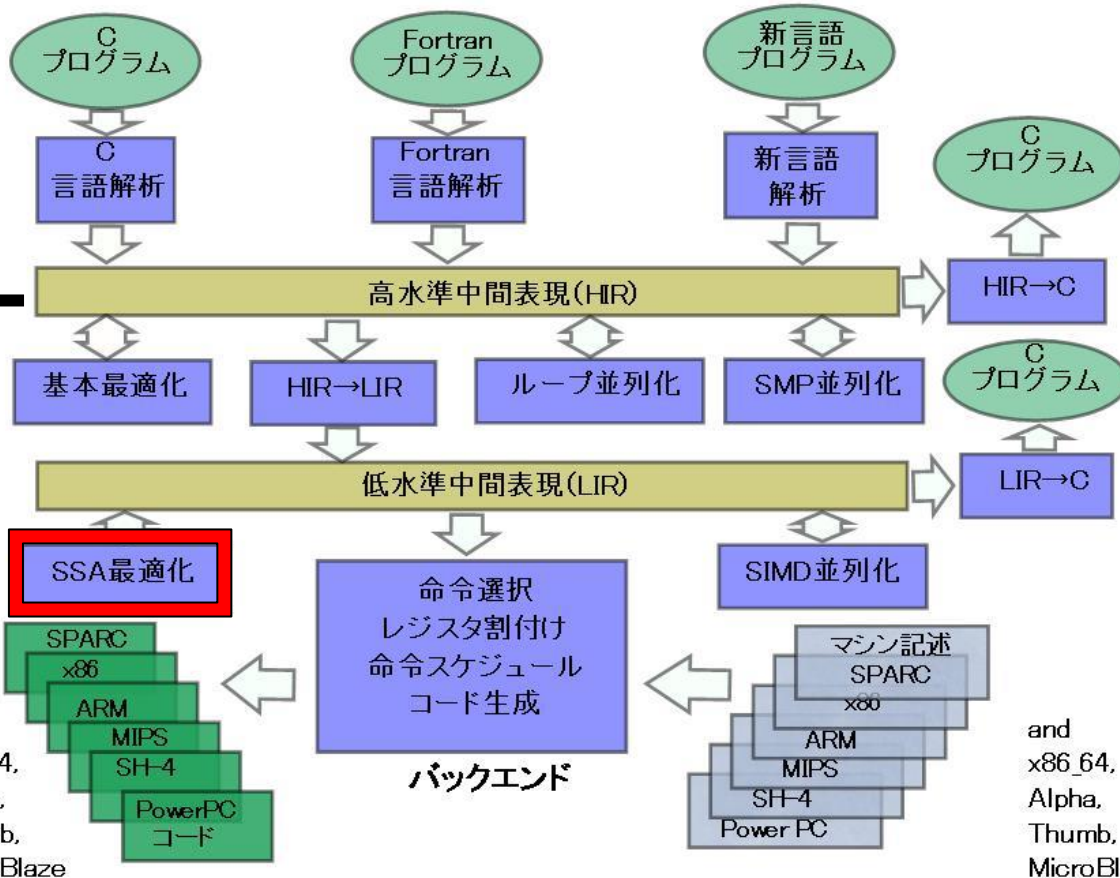
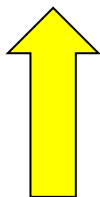


1. 字句解析
2. 構文解析
3. 意味解析
4. **コード最適化**  
(効率の良いコードへの変換)
5. コード生成

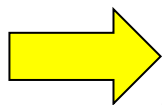


# COINSコンパイラ

字句解析  
構文解析  
意味解析



本手法





# 本研究の貢献

---

- メモリ参照の回数を低減
  - 高速な解析法に基づいて冗長なメモリ参照を除去。（3章）
    - 従来法よりも平均で約50%高速な解析法を実現
  - ループの繰返しを超えて冗長な配列参照を除去できるように拡張した手法を実現。（4章）
    - 従来法よりも実行効率を2%程度向上する。
- キャッシュメモリのミス数を低減（5章）
  - 同じ配列へのアクセスを、次元を考慮しながら連続させる。
    - ミスを30%程度低減する。



# 目次

---

1. 効率的な要求駆動型**部分冗長除去** (PRE) (3章)
  - 冗長なメモリ参照をレジスタ参照へ変換する.
2. 要求駆動型スカラ置換 (4章)
  - ループの繰返しを超えて冗長なメモリ参照をレジスタ参照へ変換する.
3. 大域ロード命令集約 (5章)
  - PREを拡張してキャッシュミス数を低減する.
4. まとめ

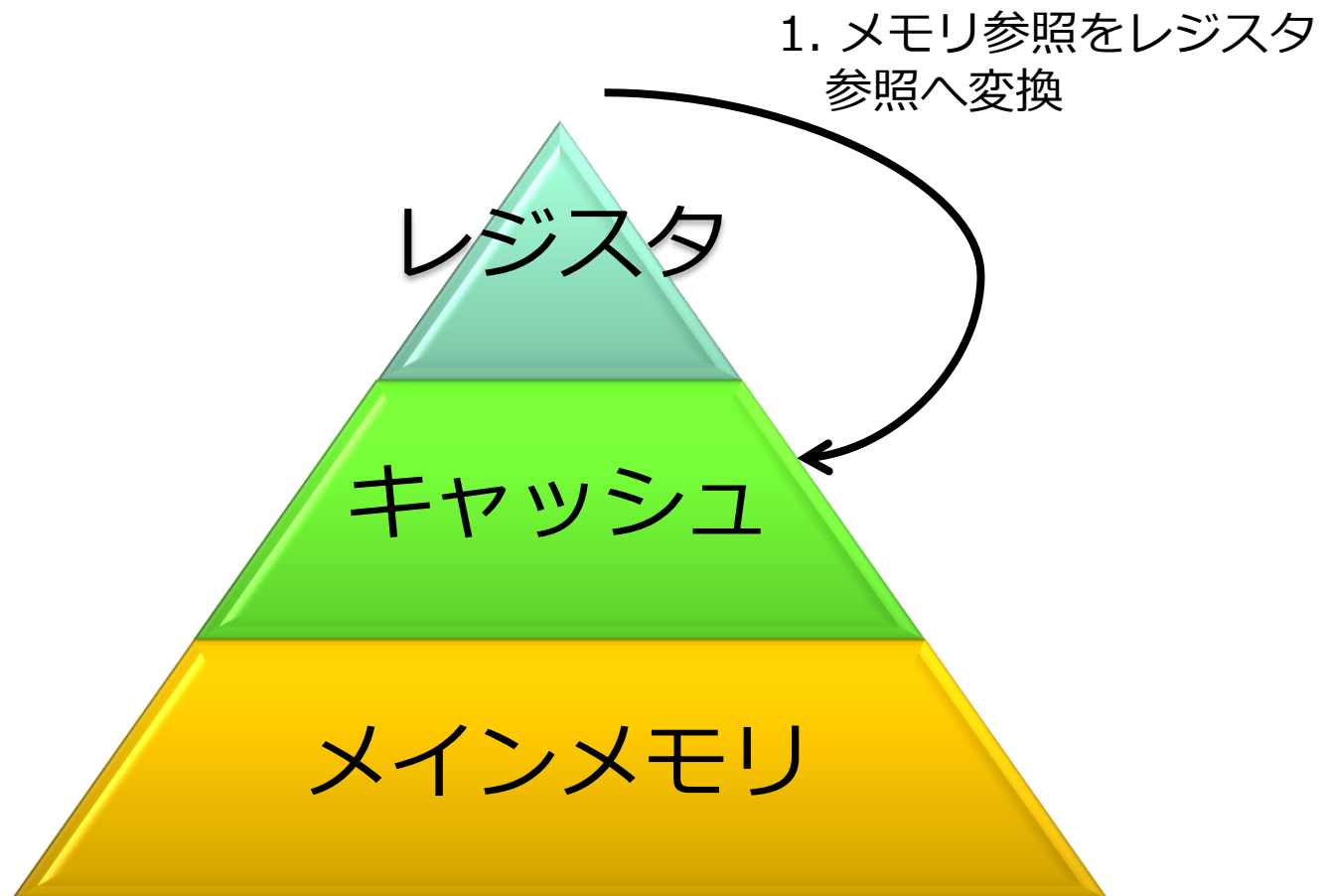


# 目次

---

1. 効率的な要求駆動型PRE（3章）
  - 冗長なメモリ参照をレジスタ参照へ変換する.
2. 要求駆動型スカラー置換（4章）
  - ループの繰返しを超えて冗長なメモリ参照をレジスタ参照へ変換する.
3. 大域ロード命令集約（5章）
  - PREを拡張してキャッシュミス数を低減する.
4. まとめ

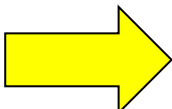
# 本手法の対象



## 冗長な式の除去

- 条件分岐が無い場合は前回の結果を参照すれば良い。

```
main() {  
    x=a[i];  
    y=a[i];  
}
```

  
a[i]を除去

```
main() {  
    x=a[i];  
    y=x;  
}
```



## 部分冗長除去 (PRE)

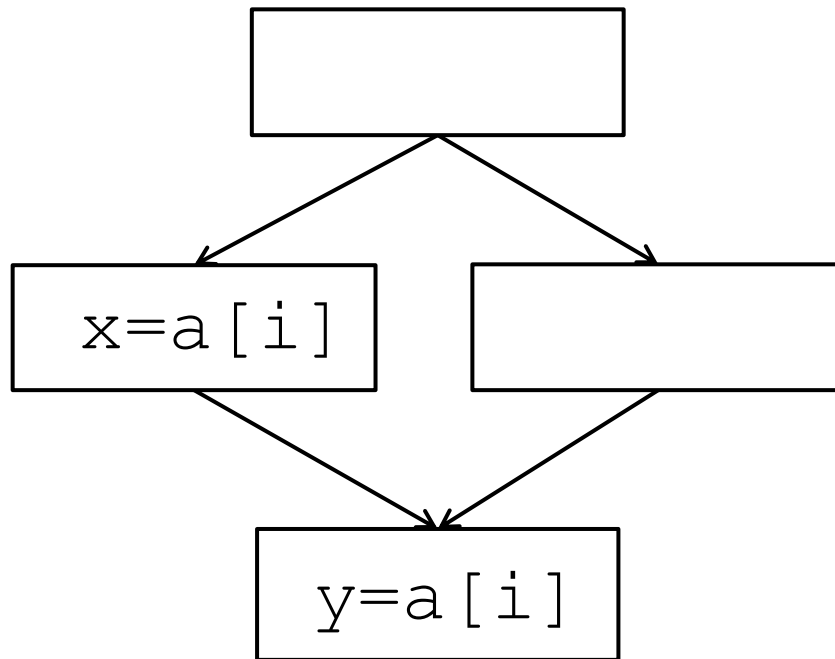
---

- 一部の実行経路上だけで冗長な式を除去する手法

```
main () {  
    if (...) {  
        x=a[i];  
    }else{  
    }  
    y=a[i];  
}
```

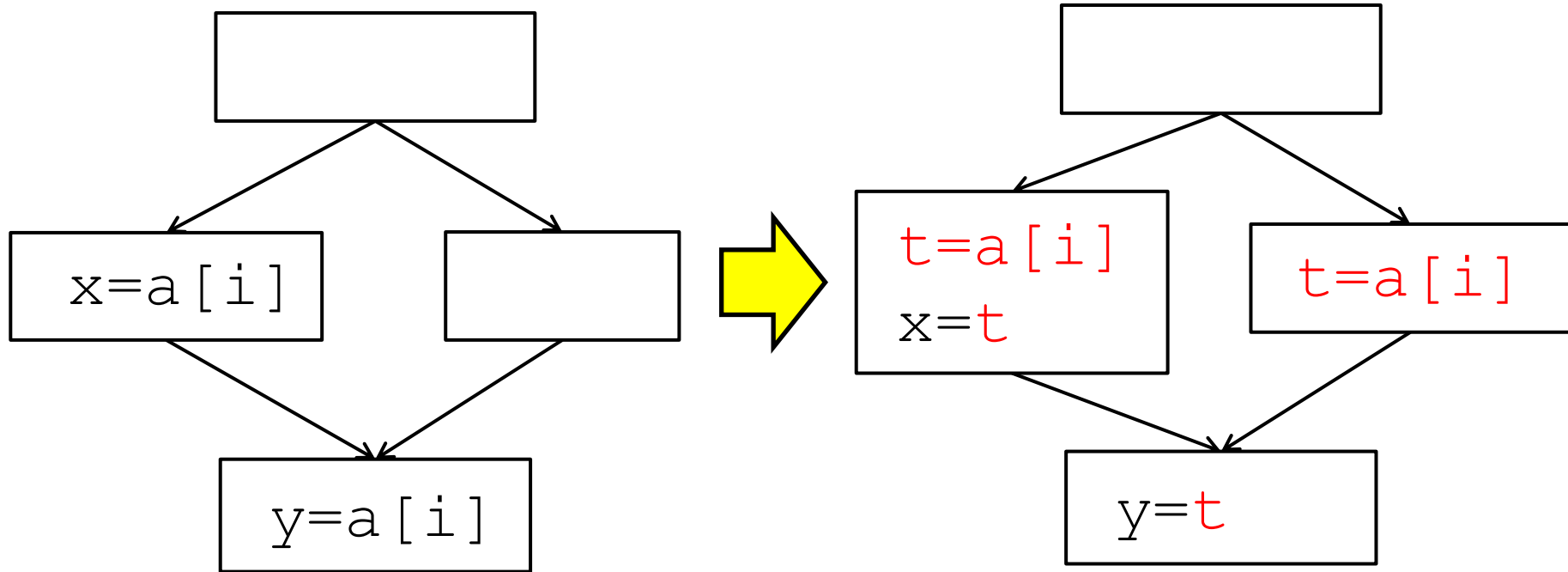
# 制御フローグラフ

- プログラムの制御の様子をグラフで表現





# PREによる冗長な式の除去の方法





## 一般的なPRE

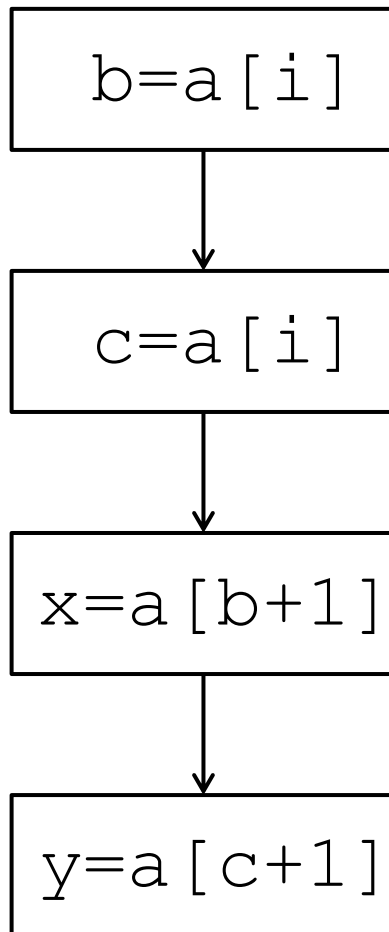
---

- 字面の一致性を用いてプログラム全体を解析.
  - 冗長性を除去した後にコピー伝播を適用する  
→ 除去できる冗長性が出現する. (副次的効果)
- ☹ PREとコピー伝播を繰返し適用する必要がある.



## 一般的なPRE

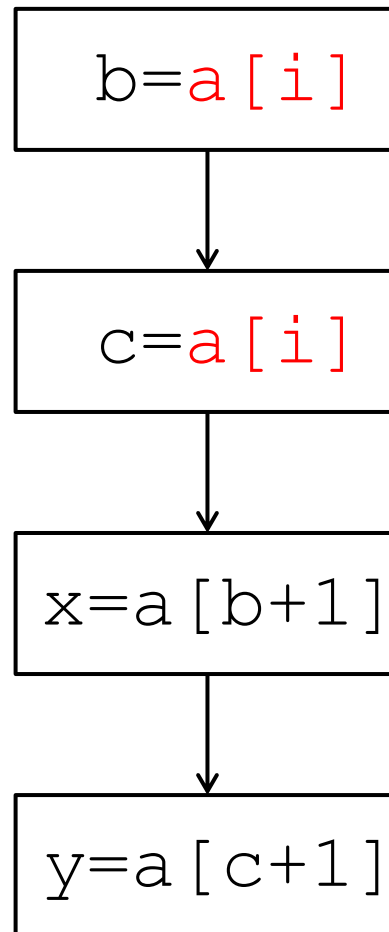
---





## 一般的なPRE

---

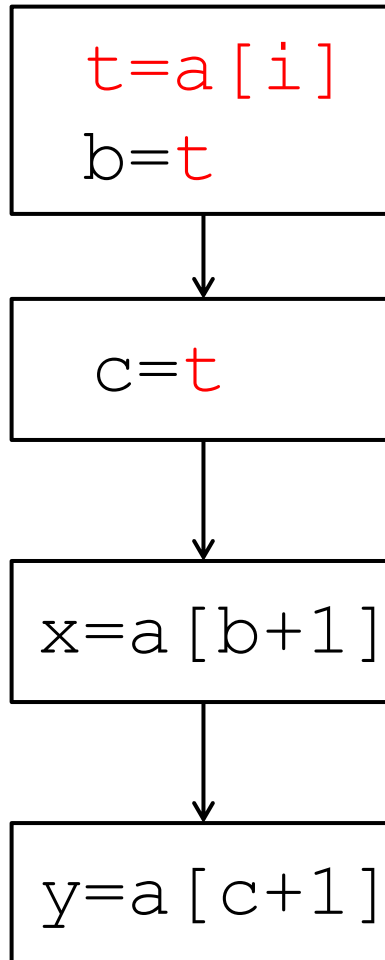


データフロー解析を用いてプログラム全体を解析



# 一般的なPRE

---

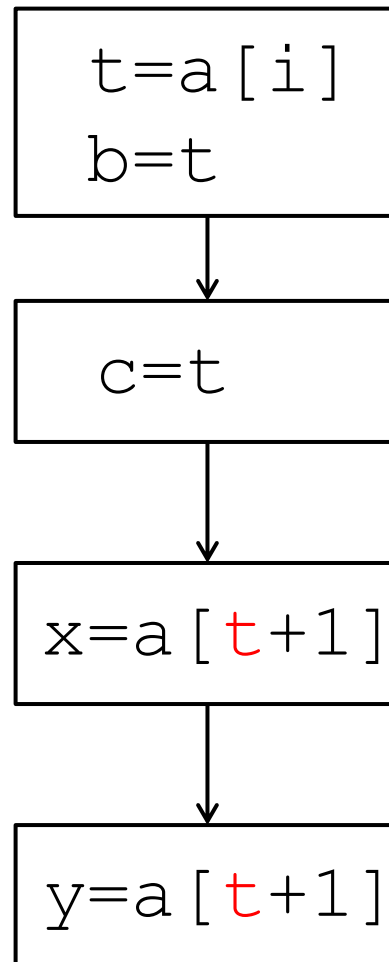


冗長性の除去



# 一般的なPRE

---

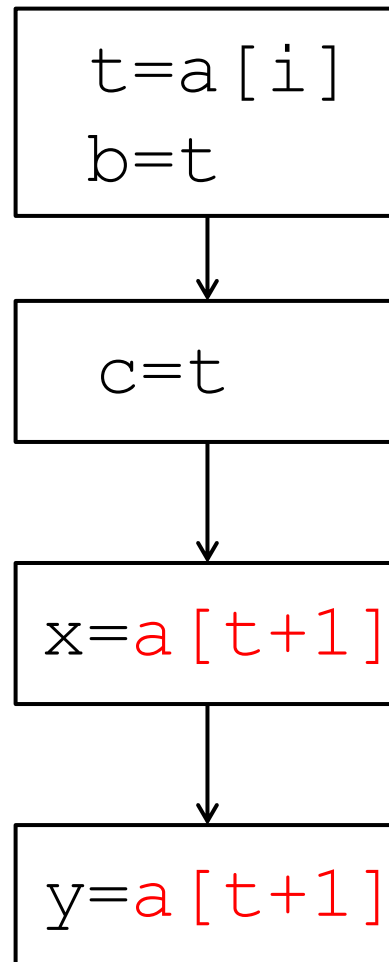


コピー伝播の適用



## 一般的なPRE

---



字面が同じ



# 効率的な要求駆動型PRE

---

- 効率的な要求駆動型PRE
  - コピー伝播を適用せずに1回の適用で副次的効果を反映
  - クエリを伝播しプログラムの限定的な範囲で冗長性を解析
  - 値番号の出現を記録し、不要なクエリの伝播を防ぐ

Sumikawa, Y. and Takimoto, M.: Effective Demand-driven Partial Redundancy Elimination, *Information Processing Society of Japan Transactions on Programming*, Vol. 6, No. 2(2013), pp. 33–44.





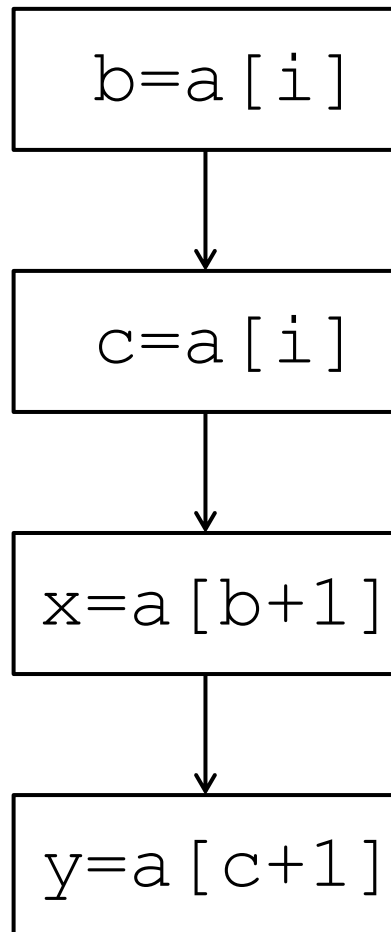
# 効率的な要求駆動型PREのアルゴリズム

---

1. 大域値番号付けを適用し異なる字面の一致性を解析  
Ex.  $a=b$ というプログラムが存在したとき,  
aとbに対して[1]という値番号を付加する.
2. 質問伝播を適用して冗長性を除去
  - 質問伝播：プログラムの解析範囲を狭めて解析効率を向上

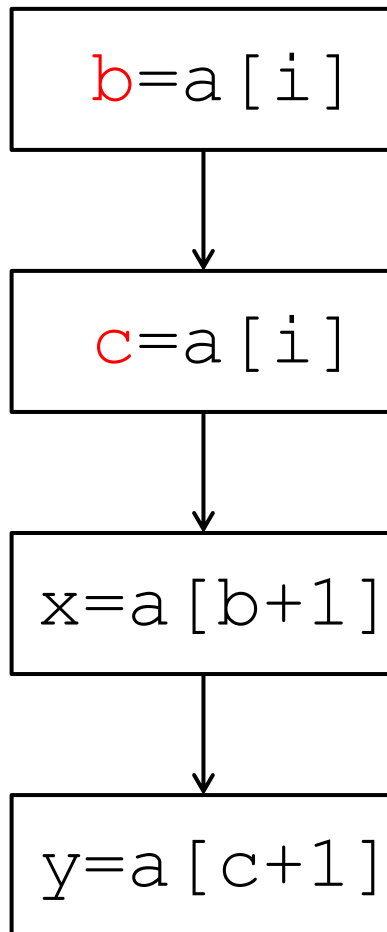


# 大域値番号付け



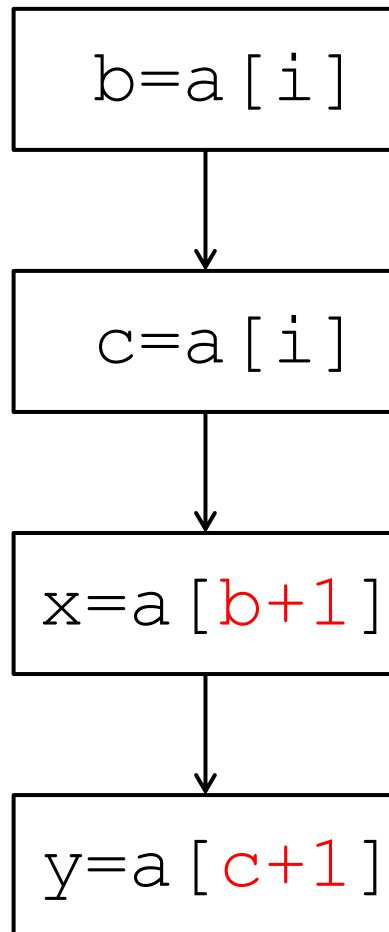
$i$	[1]
$a[i]$	[2]

# 大域値番号付け



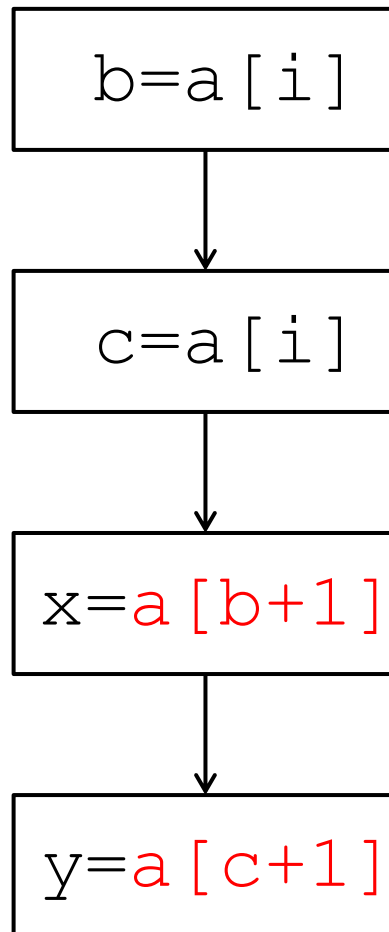
$i$	[1]
$a[i]$	[2]
$b$	[2]
$c$	[2]

# 大域値番号付け



$i$	[1]
$a[i]$	[2]
$b$	[2]
$c$	[2]
$[2]+1$	[3]

# 大域値番号付け

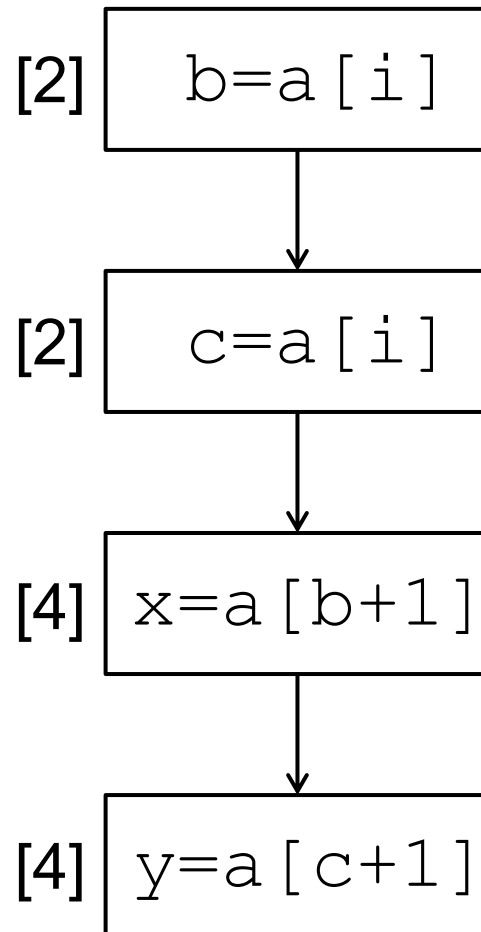


<code>i</code>	<code>[1]</code>
<code>a[i]</code>	<code>[2]</code>
<code>b</code>	<code>[2]</code>
<code>c</code>	<code>[2]</code>
<code>[2]+1</code>	<code>[3]</code>
<code>a[[2]+1]</code>	<code>[4]</code>
<code>x</code>	<code>[4]</code>
<code>y</code>	<code>[4]</code>



## 質問伝播

---



# 質問伝播

[1], [2]

[2]  $b = a[i]$

[1], [2]

[2]  $c = a[i]$

[1], [2], [3], [4]

[4]  $x = a[b+1]$

[4]  $y = a[c+1]$

# 質問伝播

[1], [2]

[2]  $b = a[i]$

[1], [2]

[2]  $c = a[i]$

[1], [2], [3], [4]

[4]  $x = a[b+1]$

[4]  $y = a[c+1]$



[4]は冗長か



# 質問伝播

[1], [2]

[2]  $b = a[i]$

[1], [2]

[2]  $c = a[i]$

[1], [2], [3], [4]

[4]  $x = a[b+1]$

[4]  $y = a[c+1]$



true

# 質問伝播

[1], [2]

[2]  $b = a[i]$

[1], [2]

[2]  $c = a[i]$

[1], [2], [3], [4]

[4]  $x = a[b+1]$

[4]  $y = x$

# 不要なクエリ伝播の防止

[1], [2]

[2]  $b = a[i]$

[1], [2]

[2]  $c = a[i]$

[1], [2], [3], [4]

[4]  $x = a[b+1]$

[7]  $y = a[c+2]$

# 不要なクエリ伝播の防止

[1], [2]

[2]  $b = a[i]$

[1], [2]

[2]  $c = a[i]$

[1], [2], [3], [4]

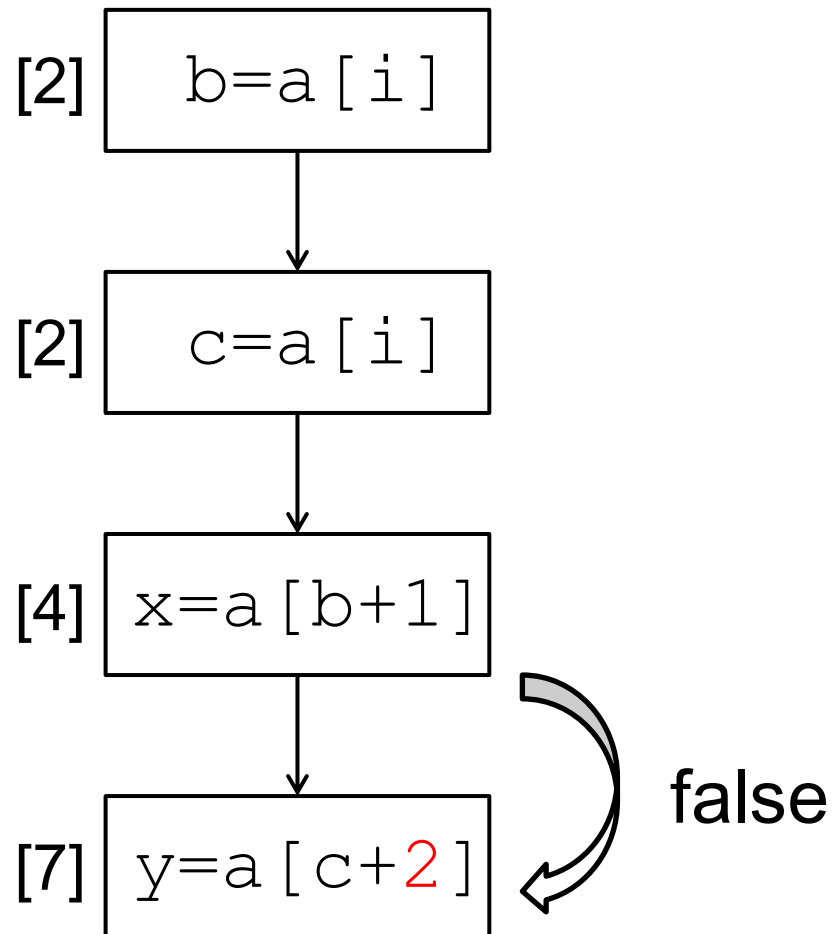
[4]  $x = a[b+1]$

3つ目の節までに  
[7]は出現しない

[7]  $y = a[c+2]$

[7]は冗長か

# 不要なクエリ伝播の防止



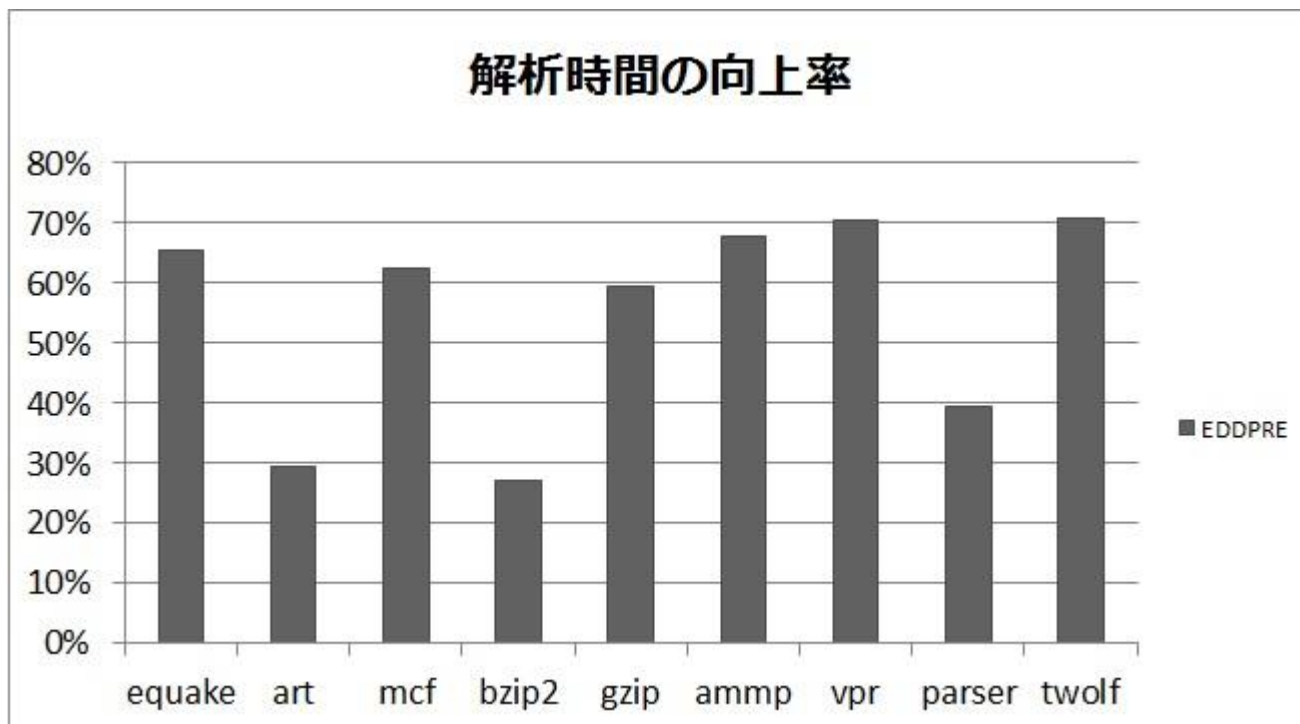


# 実験

---

- COINSコンパイラのLIR変換器として実装
- ベンチマーク
  - SPEC2000
- 解析効率（コンパイル時間）を比較
- 最適化の組み合わせ
  - PRE\*2 : 従来手法のPREを2回適用する.
  - EDDPRE : 提案手法

# 実験 | 解析効率



最大で約70%の向上



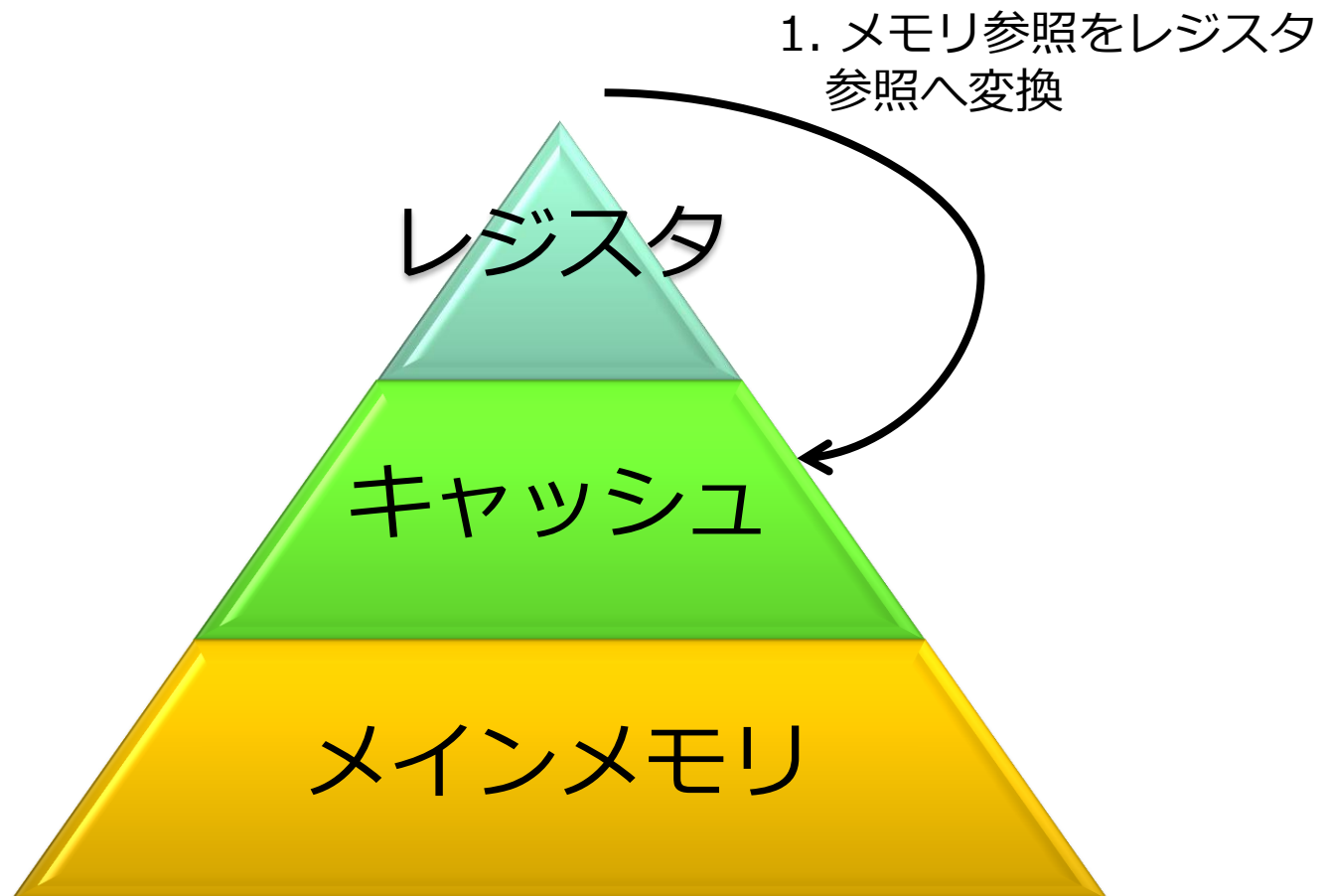
# 目次

---

1. 効率的な要求駆動型PRE (3章)
  - 冗長なメモリ参照をレジスタ参照へ変換する.
2. 要求駆動型スカラー置換 (4章)
  - ループの繰返しを超えて冗長なメモリ参照をレジスタ参照へ変換する.
3. 大域ロード命令集約 (5章)
  - PREを拡張してキャッシュミス数を低減する.
4. まとめ



# 本手法の対象





# スカラ置換のモチベーション

---

```
i=0;
while (i<10) {
    x=a[i];
    y=a[i+1];
    i++;
}
```



# スカラ置換のモチベーション

---

```
i=0;
while (i<10) {
    x=a[i];
    y=a[i+1];
    i++;
}
```

```
i=0:
    x=a[0]
    y=a[1]
```



# スカラ置換のモチベーション

---

```
i=0;
while (i<10) {
    x=a[i];
    y=a[i+1];
    i++;
}
```

```
i=0:
    x=a[0]
    y=a[1]
```

```
i=1:
    x=a[1]
    y=a[2]
```



# スカラ置換のモチベーション

---

```
i=0;  
while (i<10) {  
    x=a[i];  
    y=a[i+1];  
    i++;  
}
```

```
i=0:  
    x=a[0]  
    y=a[1]
```

```
i=1:  
    x=a[1]  
    y=a[2]
```

```
i=2:  
    x=a[2]  
    y=a[3]
```

# スカラ置換のモチベーション

```
i=0;
while (i<10) {
    x=a[i];
    y=a[i+1];
    i++;
}
```

$a[i+1]$  の値と次の繰返しの  
ときの  $a[i]$  の値が同じ

```
i=0:
    x=a[0]
    y=a[1]
    ↓
i=1:
    x=a[1]
    y=a[2]
    ↓
i=2:
    x=a[2]
    y=a[3]
```



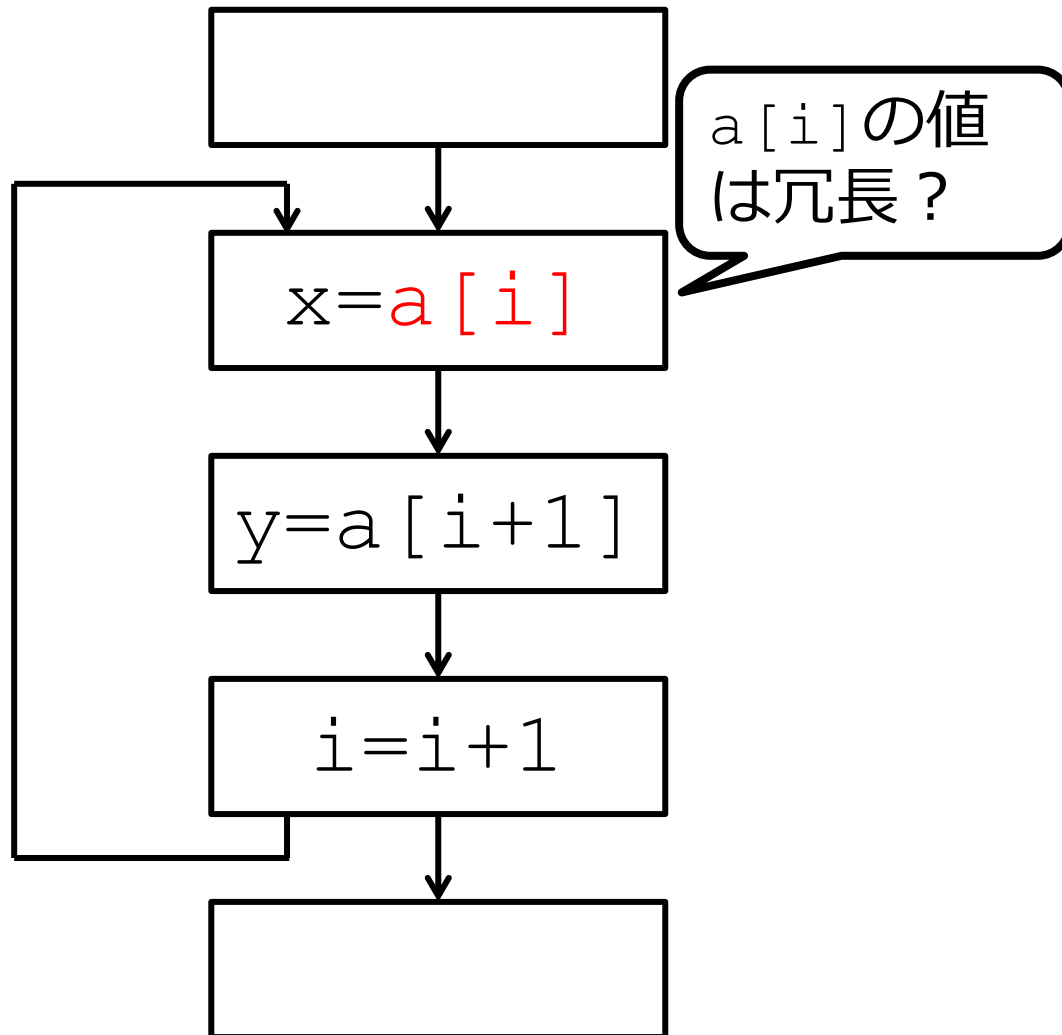
# スカラ置換

---

- 複数の繰返し内で冗長な配列参照を除去する.
  - 従来法
    - 可約なループだけが対象
    - ループ1つ1つに対して適用していた→ プログラム全体で見ると挿入した配列参照が冗長となりうる.
  - 要求駆動型スカラ置換
    - 既約なループも扱える.
    - プログラム全体から適切な挿入点を求める.

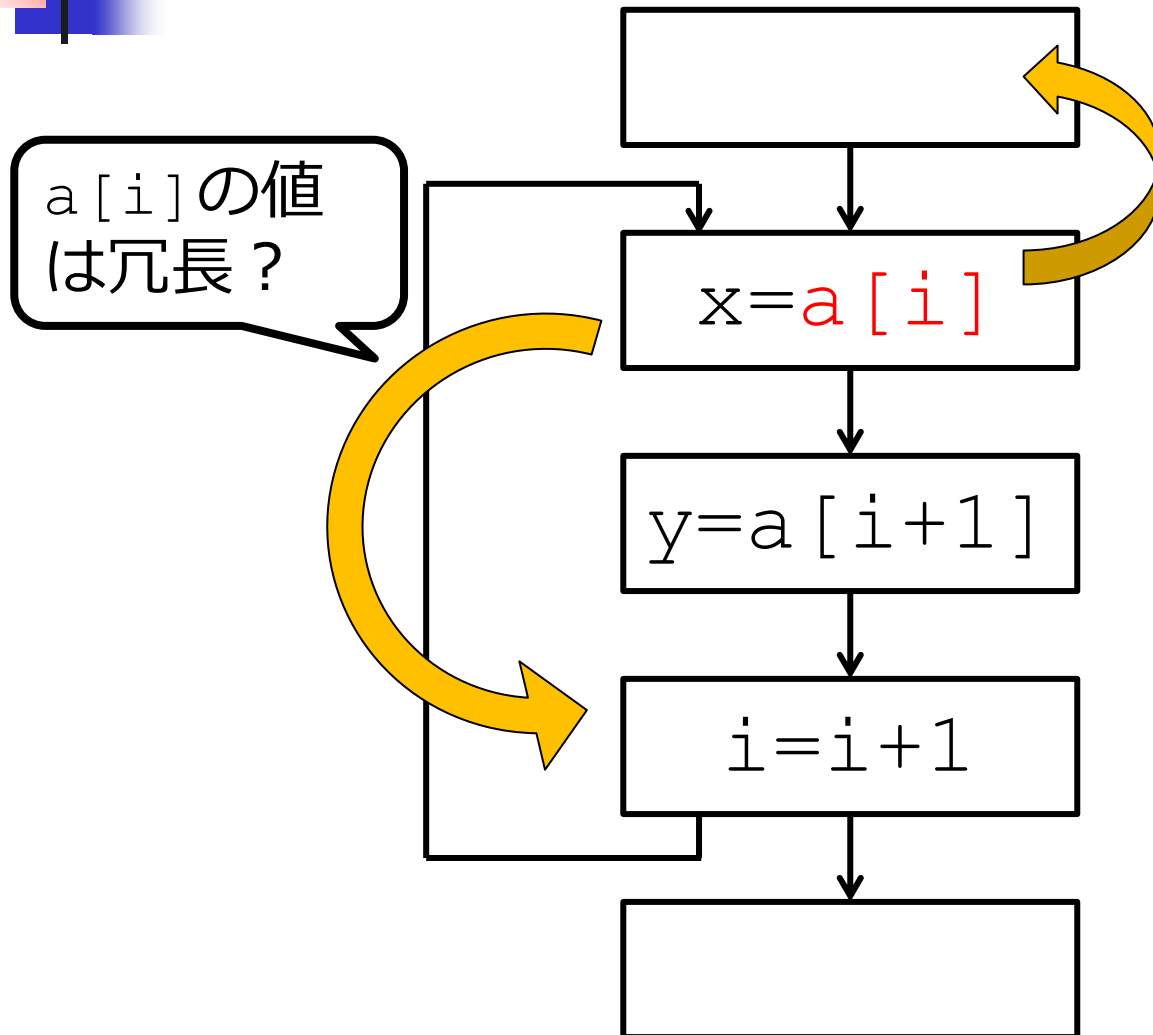
澄川靖信, 小島量, 滝本宗宏: 要求駆動型スカラ置換, コンピュータソフトウェア, 採録決定.

# 要求駆動型スカラー置換の概説

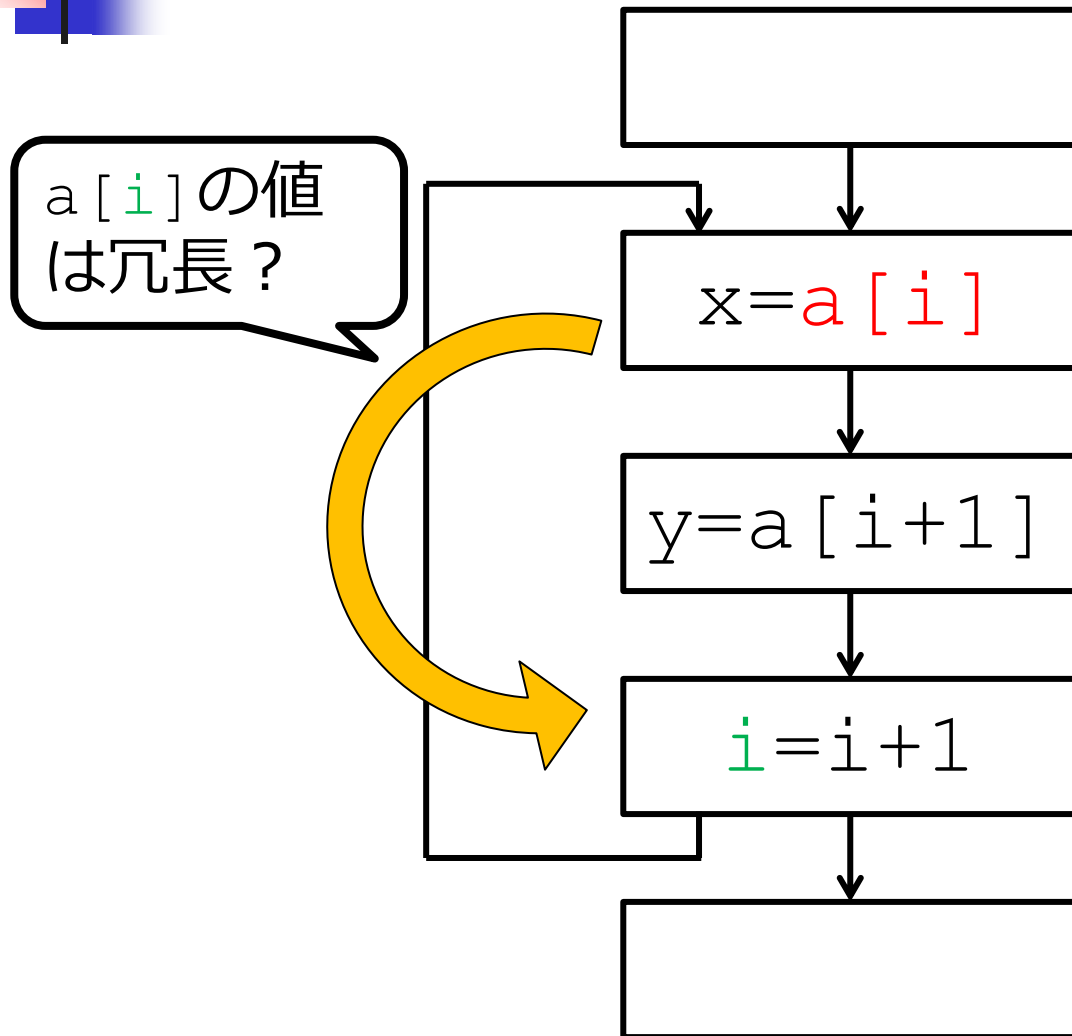




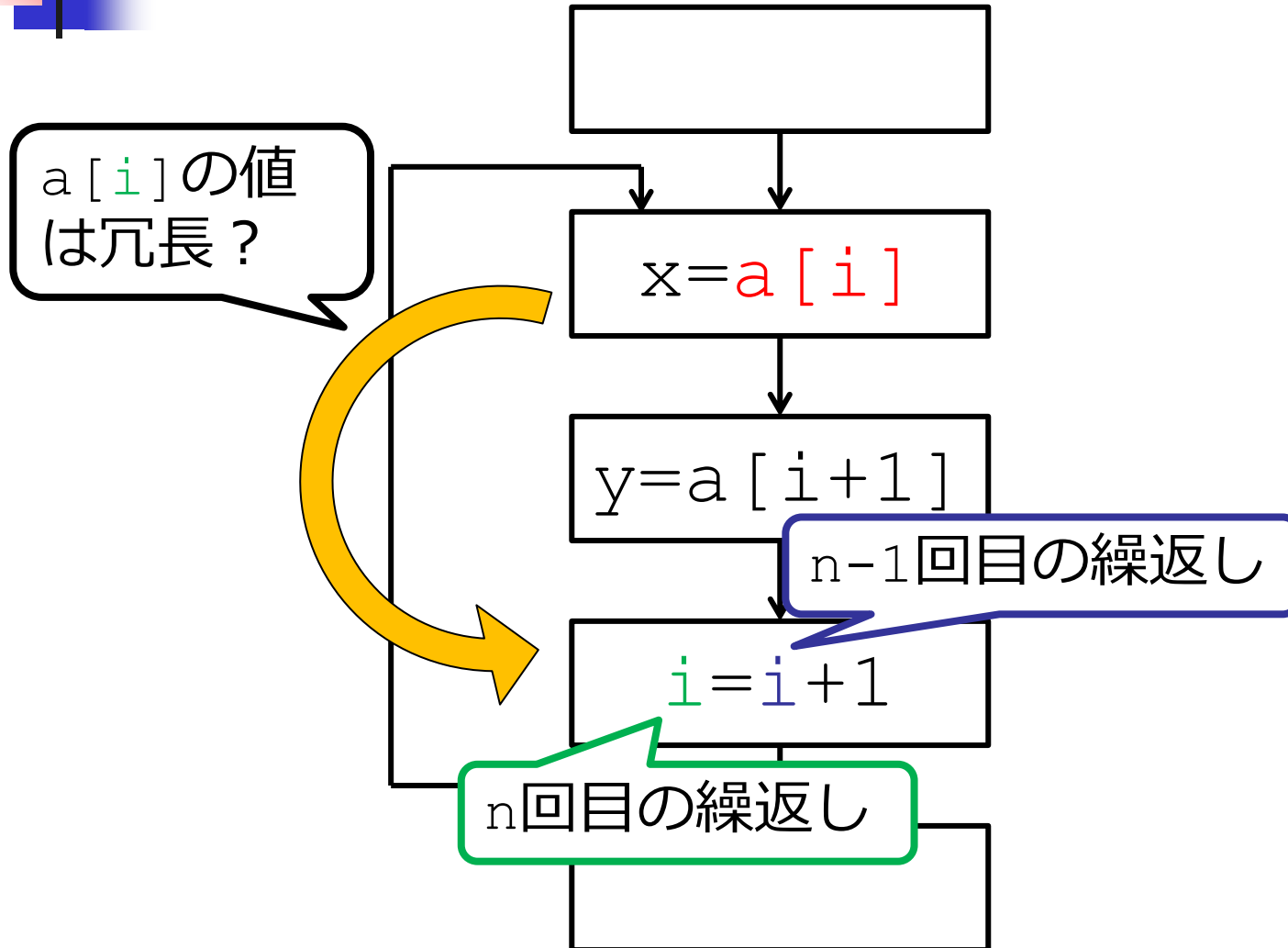
# 要求駆動型スカラー置換の概説



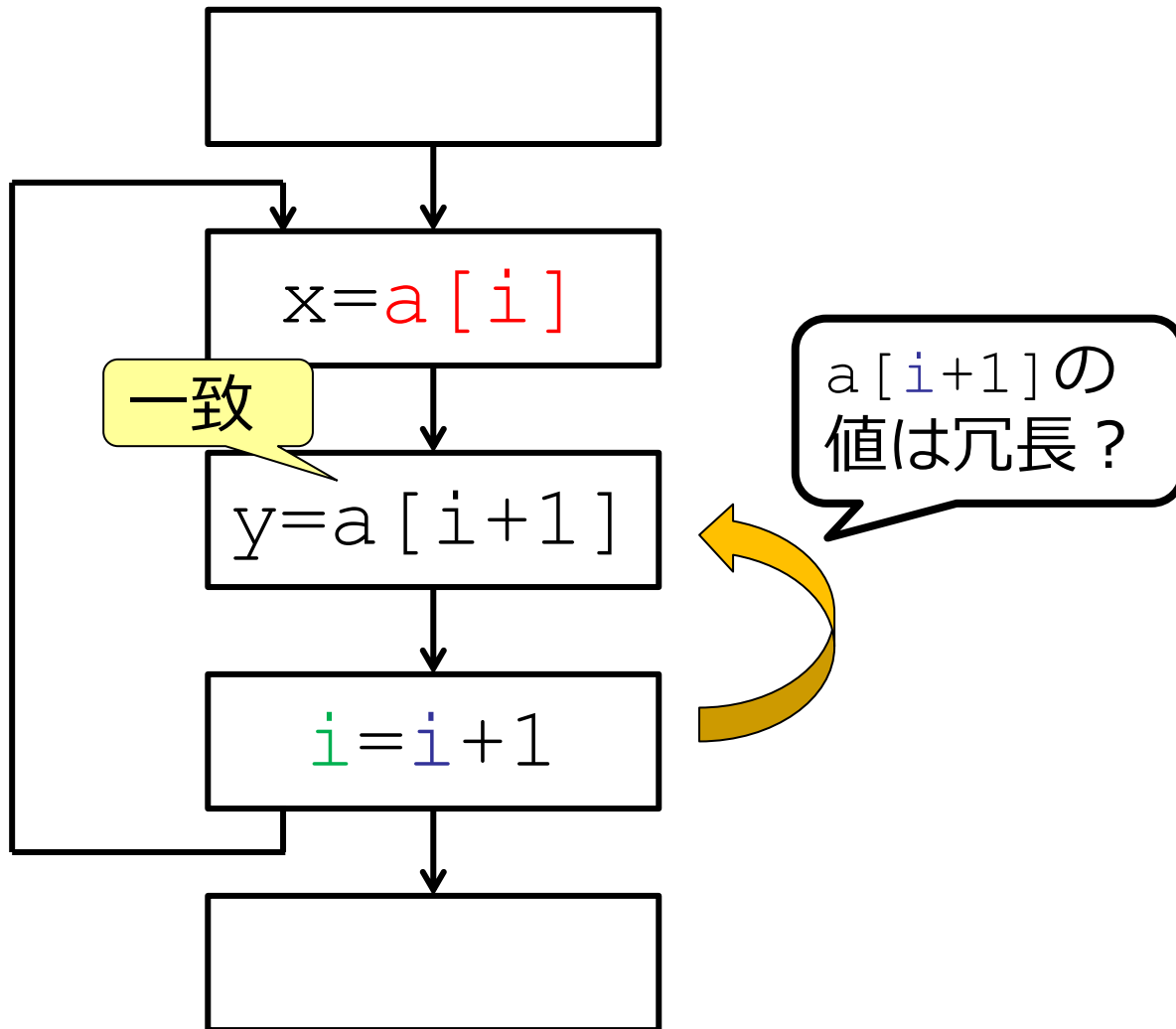
# 要求駆動型スカラー置換の概説



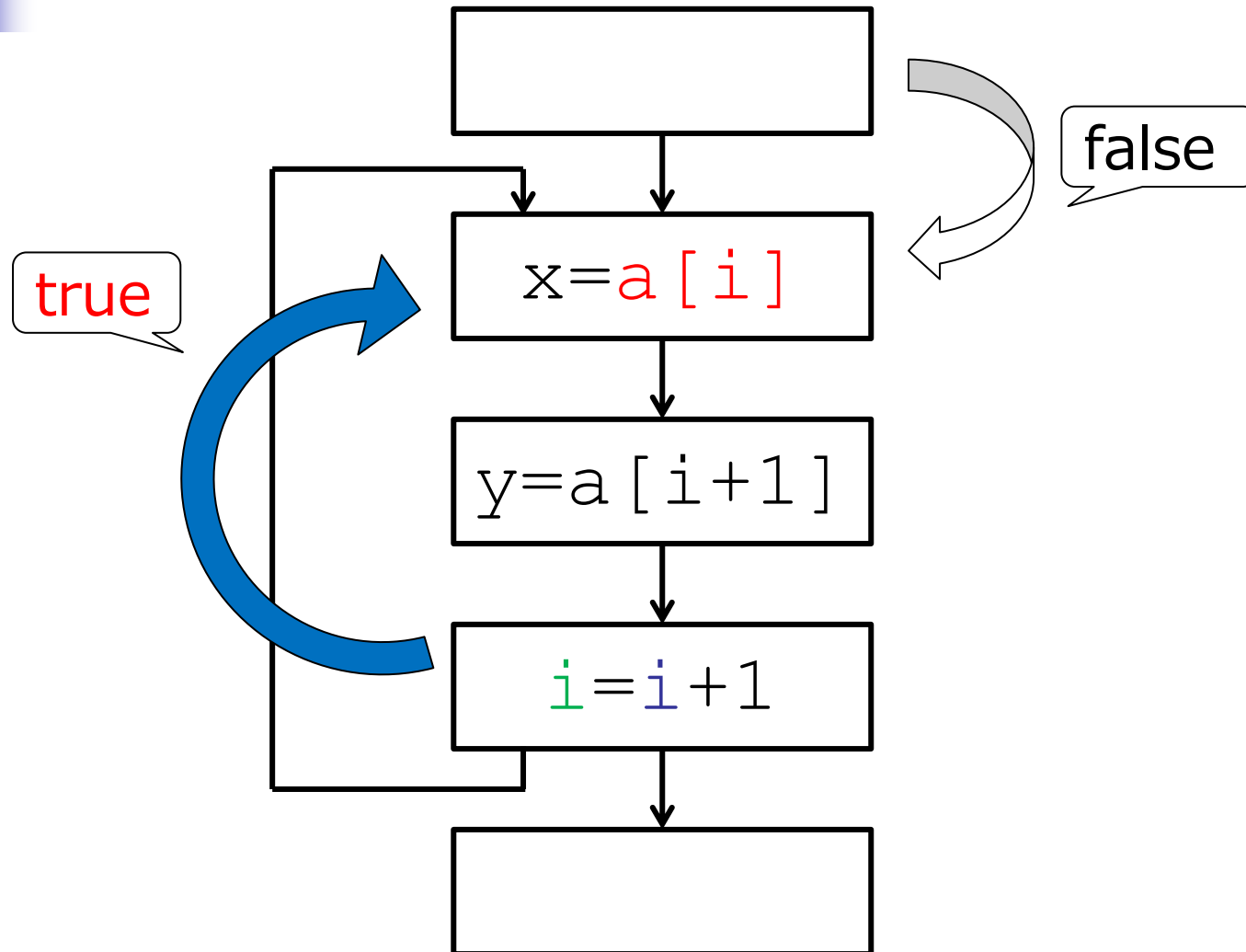
# 要求駆動型スカラー置換の概説



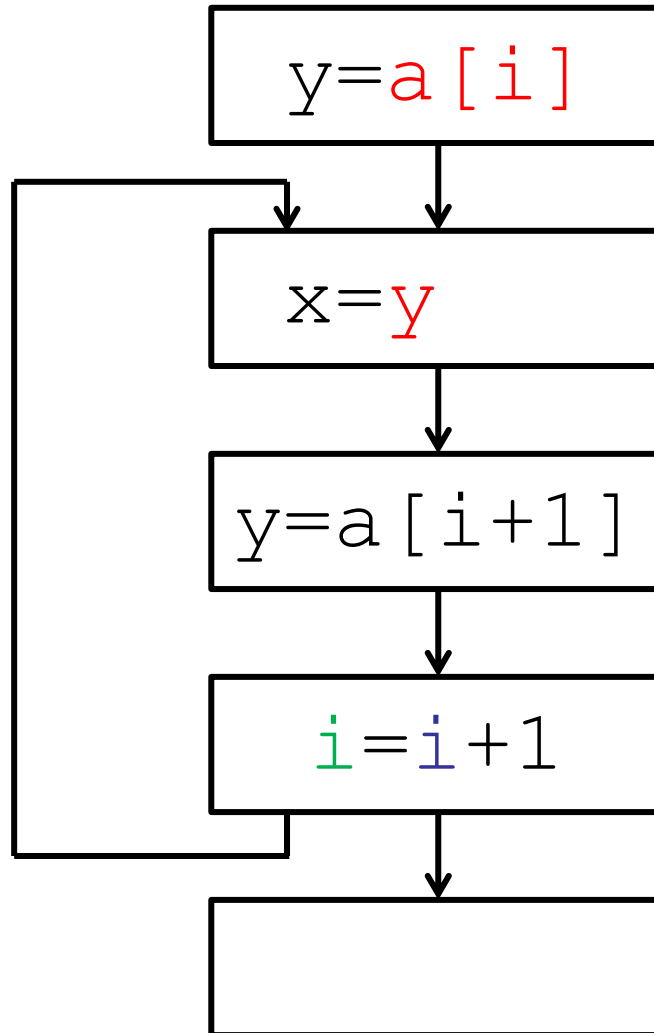
# 要求駆動型スカラー置換の概説



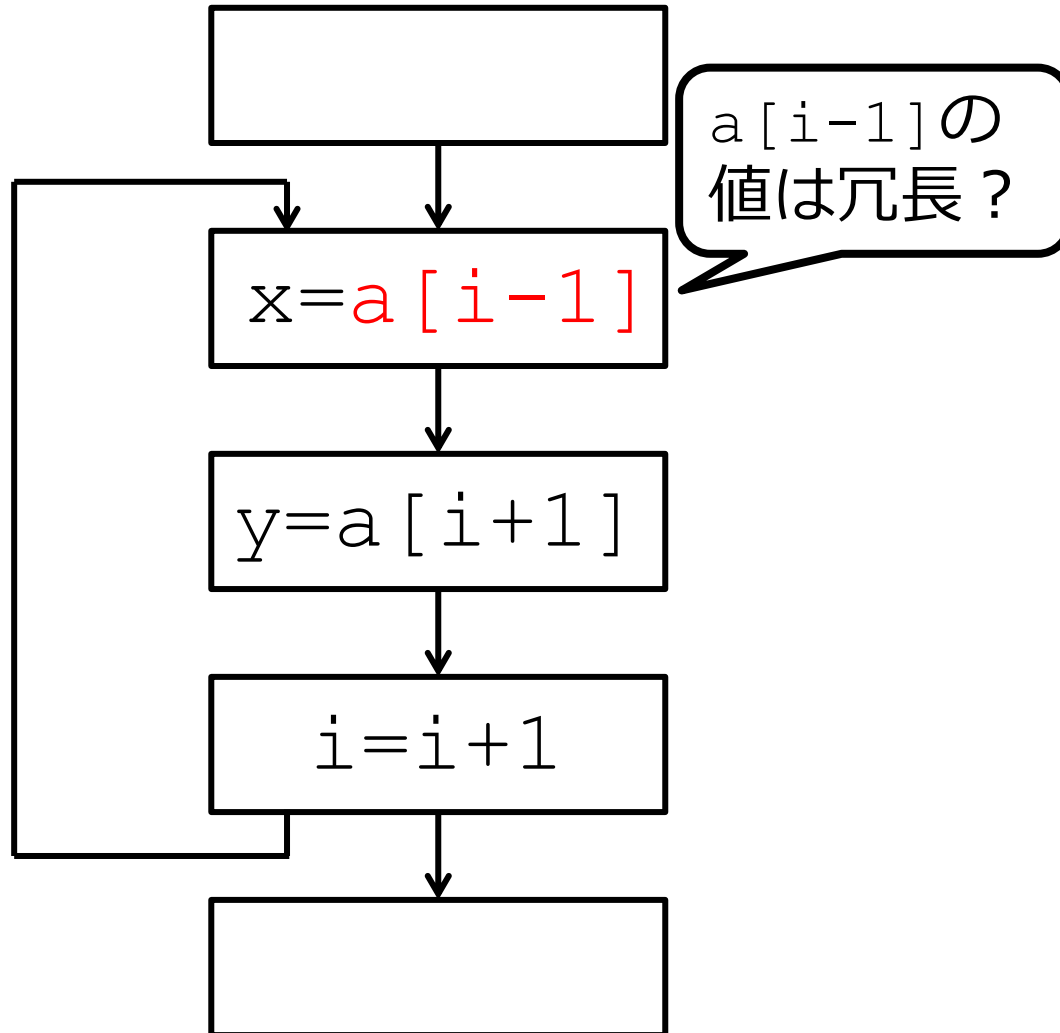
# 要求駆動型スカラー置換の概説



# 要求駆動型スカラー置換の概説



# 要求駆動型スカラー置換の概説





# 実験結果

---

## ■ 環境

- コンパイラ : COINS (Cコンパイラ)
- CPU : Xeon E5-1660 3.3GHz
- OS: Debian 64bit
- ベンチマーク : SPEC2000
  - 各プログラムのループだけを使用した.

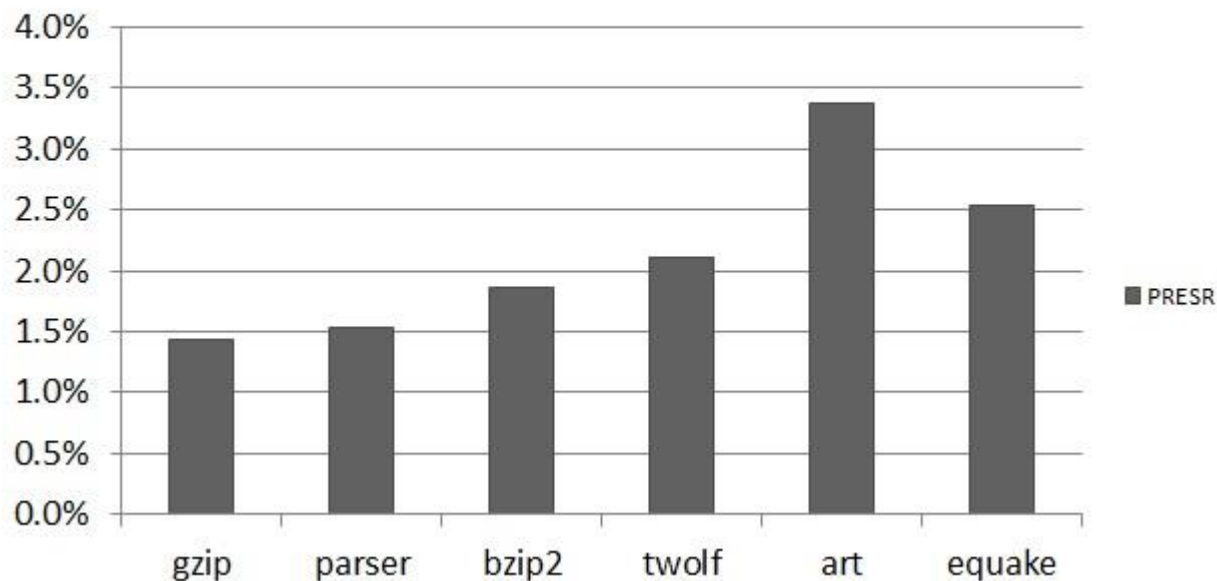
## ■ 最適化

- TSR+: 一般的なスカラー置換の従来法
- PRESR: 提案手法, 3回前の繰返しまでを解析



# 実験結果 | 実行効率 (秒)

## 目的コードの実行効率の向上率



最大で約3.4%の向上



# 目次

---

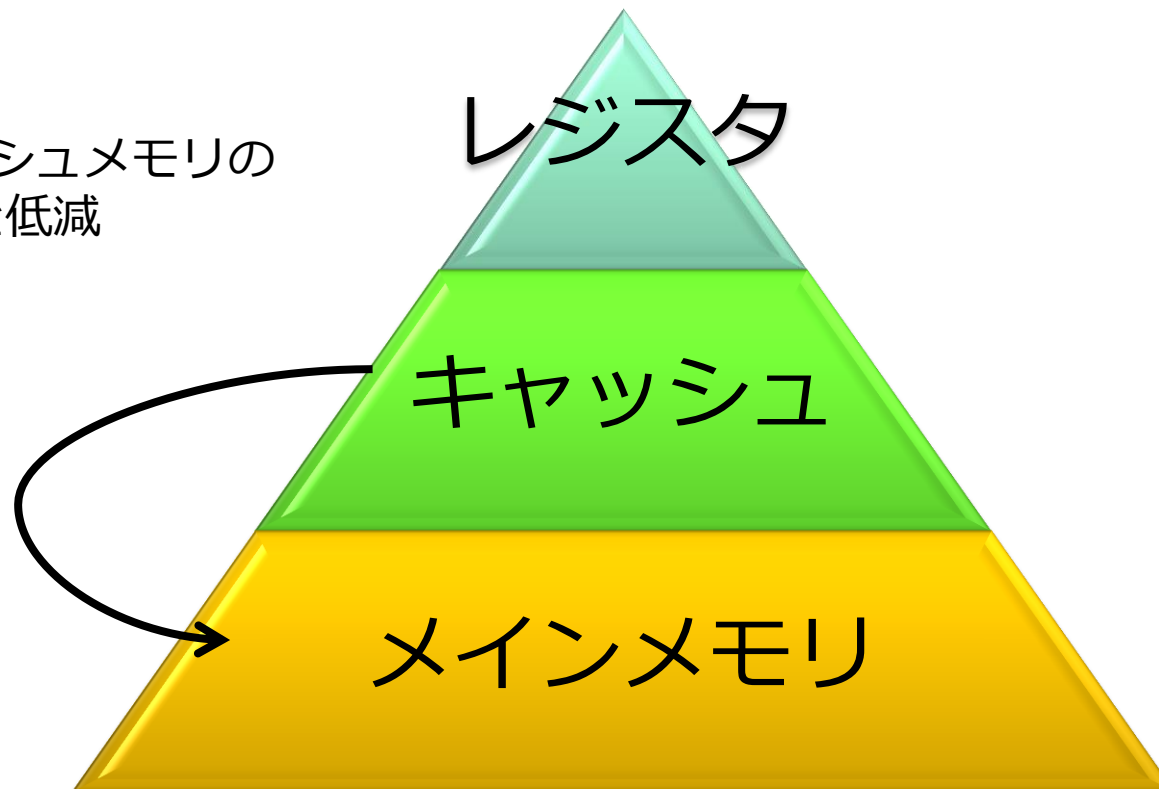
1. 効率的な要求駆動型PRE (3章)
  - 冗長なメモリ参照をレジスタ参照へ変換する.
2. 要求駆動型スカラー置換 (4章)
  - メモリ参照をレジスタ参照へ変換する.
3. 大域ロード命令集約 (5章)
  - PREを拡張してキャッシュミス数を低減する.
4. まとめ



## 本手法の対象

---

2. キャッシュメモリの  
ミス数を低減





# 大域ロード命令集約

---

- 大域ロード命令集約
  - メモリ参照の局所性を効率良く利用できるようにPREを拡張  
→ キャッシュミス数を低減する.

Sumikawa, Y., Takimoto, M.: Global load instruction aggregation based on code motion. PAAP '12, IEEE Computer Society, pp.149–156

# モチベーション

```
main () {  
    ... = a[i]  
    ... = b[i]  
    ... = a[i+1]  
}
```

00		
01		
10		
11		

キャッシュ

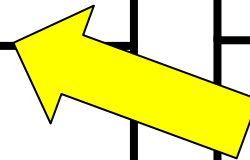
b[i]	0000
b[i+1]	0001
⋮	
a[i]	1000
a[i+1]	1001
⋮	

メインメモリ

# モチベーション

```
main () {  
    ... = a[i]  
    ... = b[i]  
    ... = a[i+1]  
}
```

00	a[i]	a[i+1]
01		
10		



b[i]	0000
b[i+1]	0001
⋮	
⋮	
a[i]	1000
a[i+1]	1001

**1000**<sub>(2)</sub> (10進数: 8) をライン数 (4) で割った余りの番地に配置する.

# モチベーション

```
main () {  
    ...=a[i]  
    ...=b[i]  
    ...=a[i+1]  
}
```

00	a[i]	a[i+1]
01		
10		

b[i]	0000
b[i+1]	0001
⋮	
a[i]	1000
a[i+1]	1001
⋮	



メインメモリ

# モチベーション

```
main () {  
    ... = a[i]  
    ... = b[i]  
    ... = a[i+1]  
}
```

00	b[i]	b[i+1]
01		
10		
11		

キャッシュ

b[i]	0000
b[i+1]	0001
⋮	
a[i]	1000
a[i+1]	1001
⋮	

メインメモリ



# モチベーション

```
main () {  
    ...=a[i]  
    ...=b[i]  
    ...=a[i+1]  
}
```

00	b[i]	b[i+1]
01		
10		

b[i]	0000
b[i+1]	0001
⋮	
a[i]	1000
a[i+1]	1001
⋮	

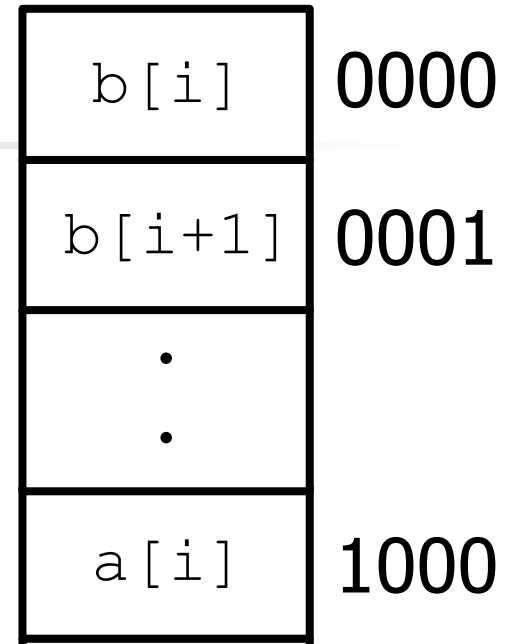
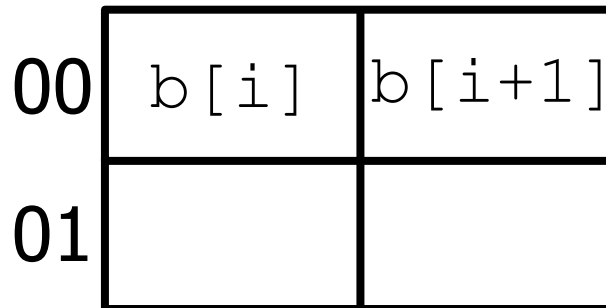


メインメモリ

# モチベーション

同じ配列のデータを同時に配置

```
main() {  
    ... = a[i]  
    ... = b[i]  
    ... = a[i+1]  
}
```



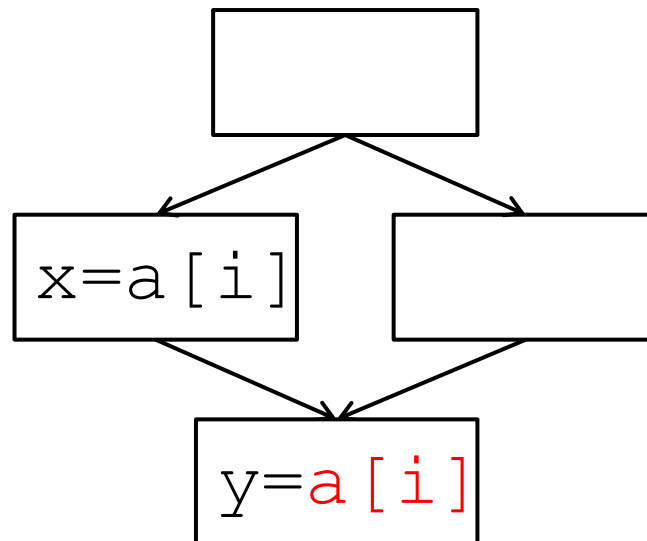
同じ配列へのアクセスを連続するよう式の順序を入れ替えば良い。

キャッシュ

メインメモリ

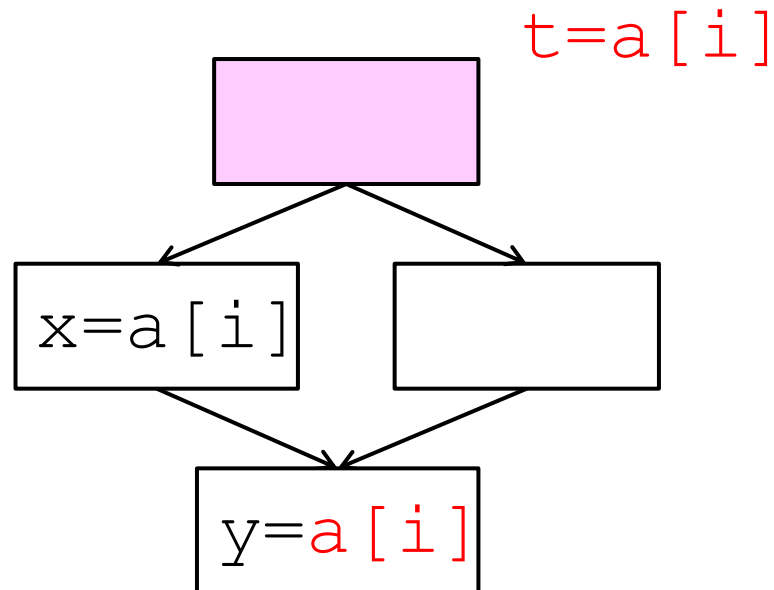
# 大域ロード命令集約の実現

- PREの1つの実現方法で**怠けたコード移動**を拡張



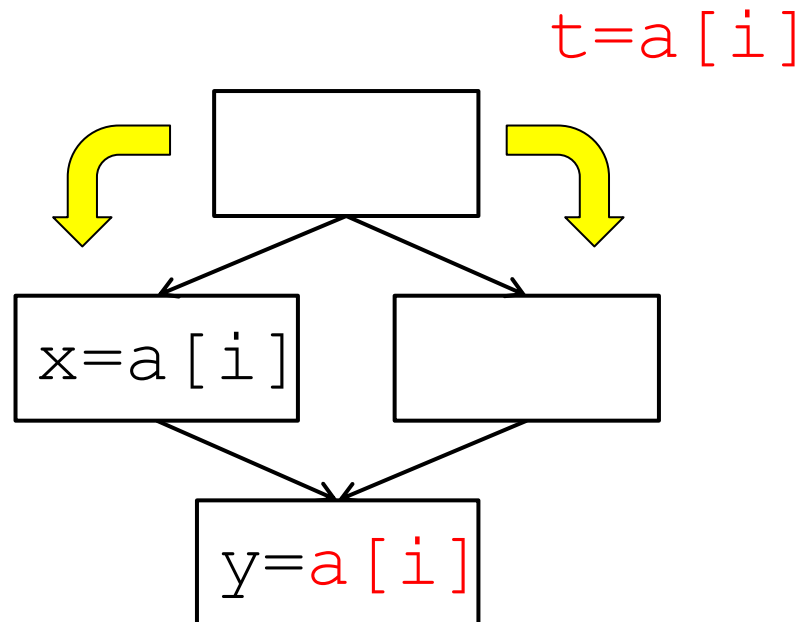
# 怠けたコード移動のアルゴリズム

1. 式を挿入できる最も開始節に近い節を求める.



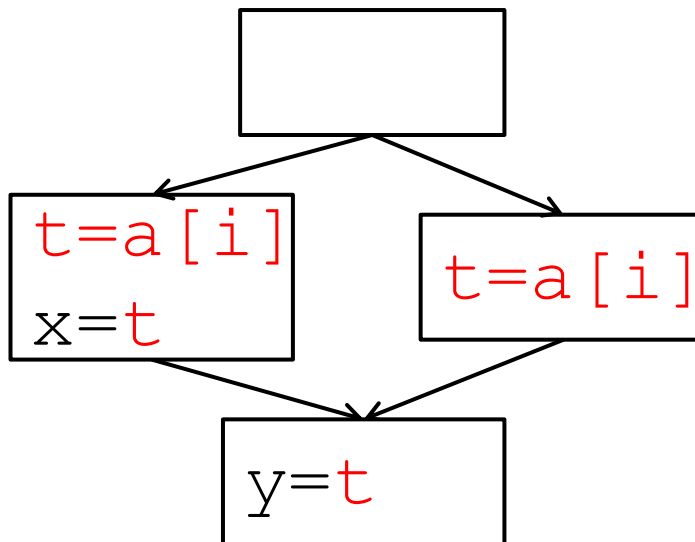
# 怠けたコード移動のアルゴリズム

2. 式の挿入を可能な限り遅らせる.



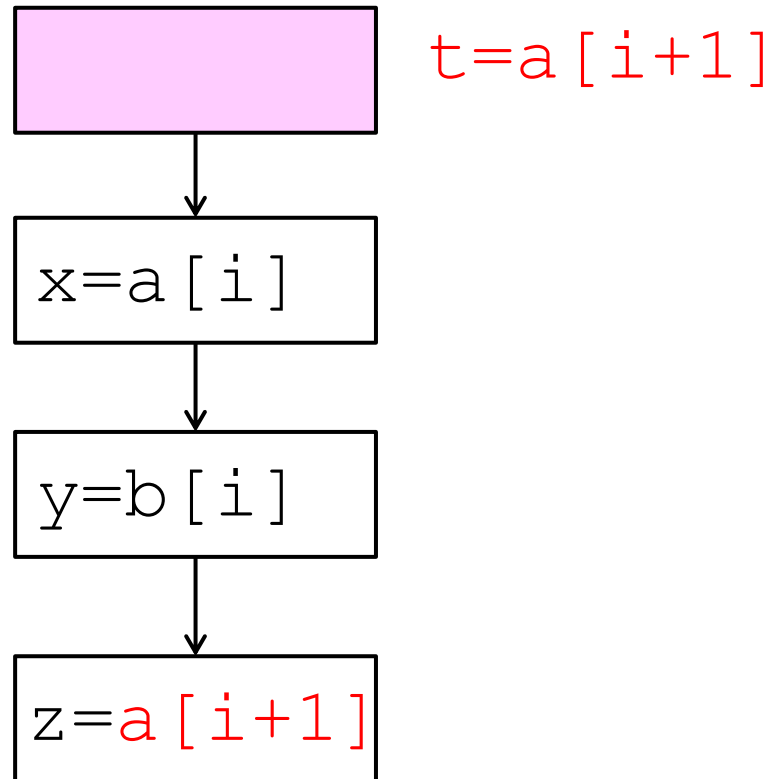
# 怠けたコード移動のアルゴリズム

3. 式を挿入して冗長性を除去する.



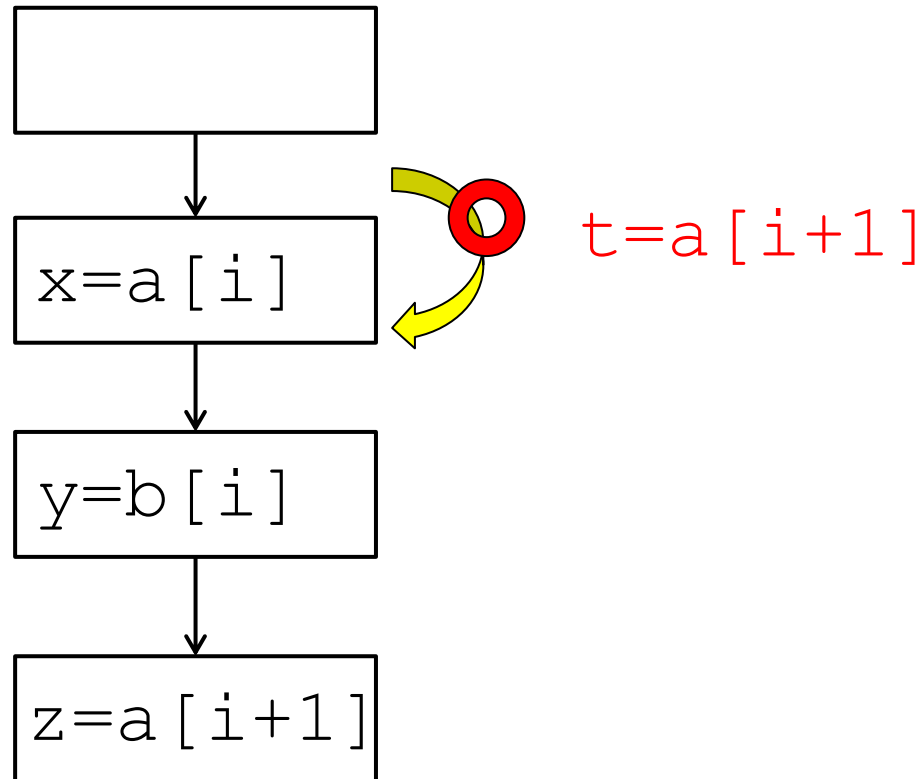
# 大域ロード命令集約のアルゴリズム

1. 式を挿入できる最も開始節に近い節を求める.



# 大域ロード命令集約のアルゴリズム

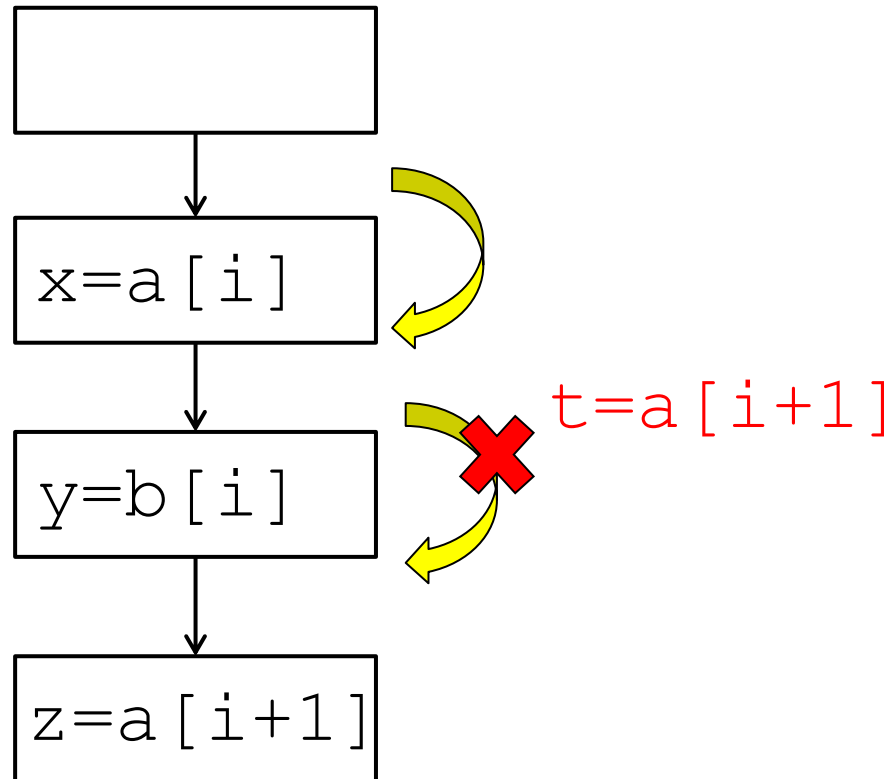
2. 式の挿入を可能な限り遅らせる.





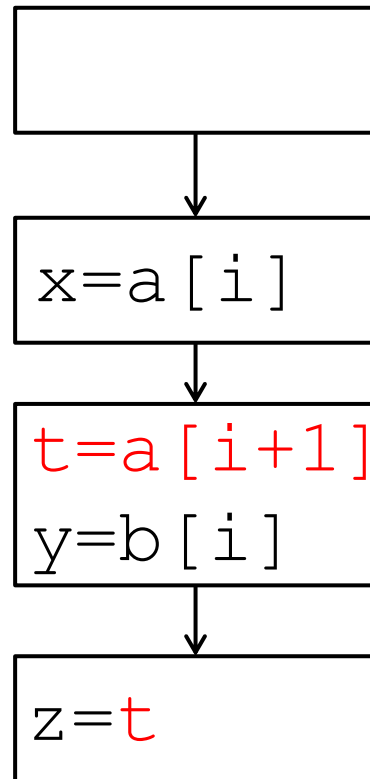
# 大域ロード命令集約のアルゴリズム

2. 式の挿入を可能な限り遅らせる.



# 大域ロード命令集約のアルゴリズム

3. 式を挿入して冗長性を除去する.





## 多次元配列への拡張

---

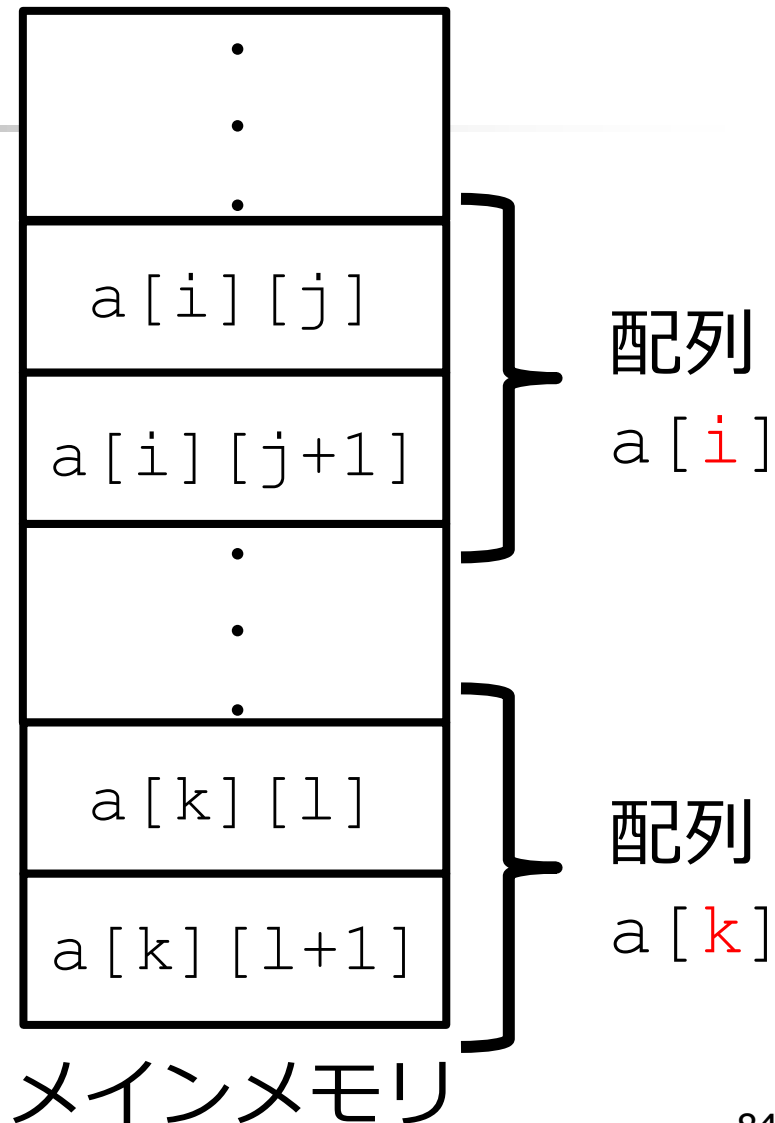
- 比較的サイズが大きい配列の場合
  - 同じ配列への参照を連続させても効果が無い場合がある.
- 多次元配列
  - 上位次元がその下位次元の配列とみなせる.

Sumikawa, Y., Takimoto, M.: Global Load Instruction Aggregation Based on Array Dimensionsload. PAAP '14, IEEE Computer Society, pp.123–129

## 多次元配列への拡張

```
main () {  
    ... = a[i][j]  
    ... = a[k][l]  
    ... = a[i][j+1]  
}
```

※ C言語を仮定したときの  
メモリレイアウト。





# 実験結果

---

## ■ 環境

- コンパイラ : COINS
- CPU : core2duo 1.6GHz
- OS: CentOS
- ベンチマーク : SPEC2000

## ■ 最適化

- LCM-MEM : 怠けたコード移動
- GLIA: 大域ロード命令集約
- MDGLIA: 次元を考慮した大域ロード命令集約

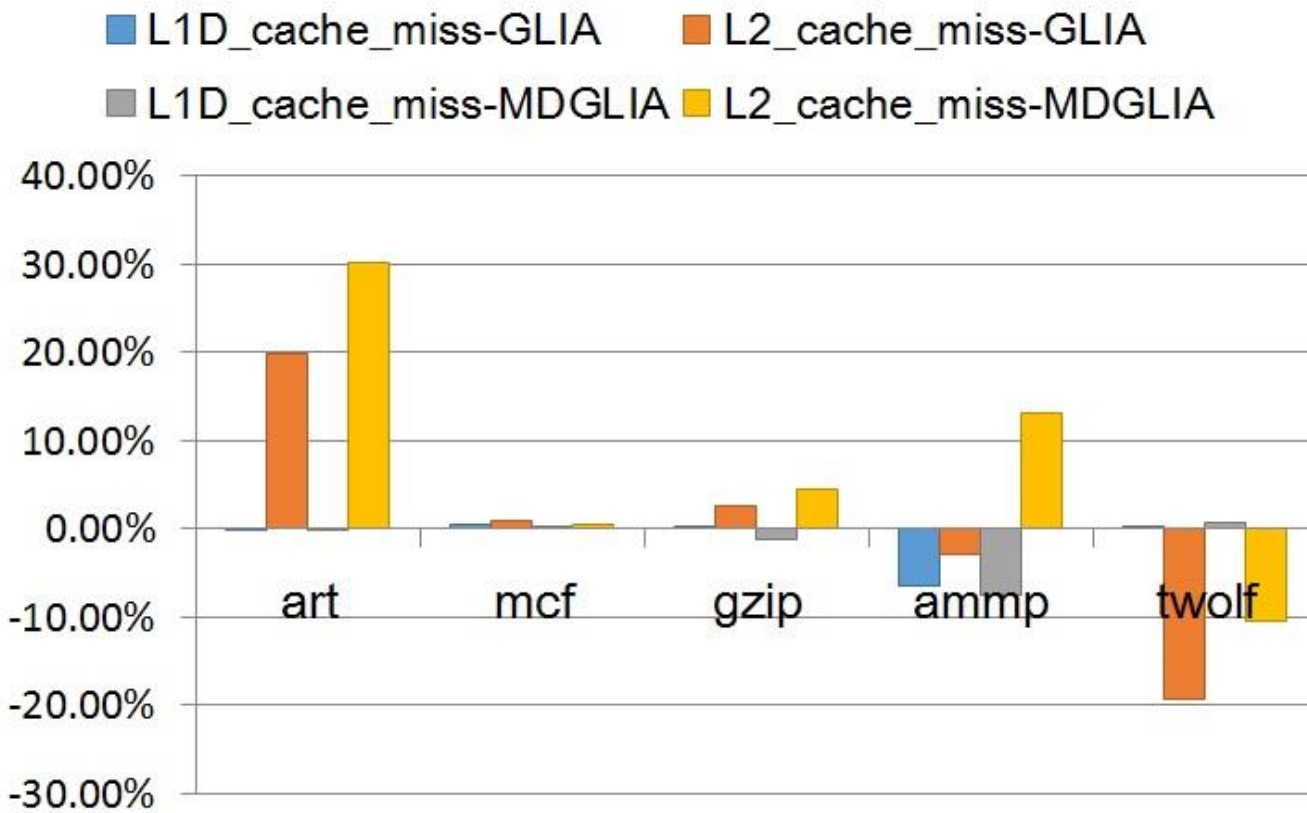


# キャッシュメモリのパラメータ

---

パラメータ	L1D	L2
総サイズ (KB)	32	3,072
ラインサイズ (bytes)	64	64
ライン数	512	49,152
連想度	8	12

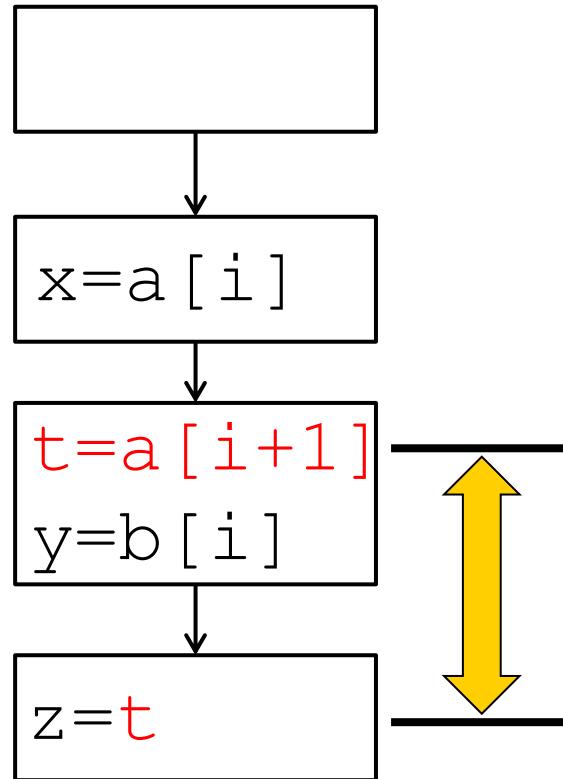
# 実験結果 | キャッシュミス数



最大で約30%の向上

# キャッシュミス増加の原因

## レジスタスピル増加



`a[i+1]` のデータを  
レジスタが保持す  
る期間の増分





## まとめ | メモリ階層最適化の提案

---

- メモリ参照のレジスタ参照への変換
  - 従来法より解析効率が約50%良いPREの提案（3章）
  - プログラム全体の中で適切な挿入点を求め、繰返しを越えて冗長となる配列参照を除去するPRESRの提案（4章）
    - 従来法よりも平均で約2%，最大で約3.4%の実行効率の向上
- キャッシュミス低減によるメインメモリ参照の低減
  - 同じ配列へのアクセスを連続にするGLIAの提案（5章）
  - 多次元配列の集約行うMDGLIAの提案（5章）
    - L2キャッシュミスを最大で30%低減.

コード移動によるメモリ階層最適化の枠組みを与えた