



Minimizing Memory Access By Improving Register Usage Through High-Level Transformations

Shan Li

School of Computer Engineering
Nanyang Technological University
Singapore 639798

Mohammed Javed Absar (Speaker)

STMicroelectronics Asia Pacific Pte. Ltd.
Singapore 117674

Contents

- Introduction
 - Project Goal
 - Means to reach the Goal
 - Contributions
- Multiple-index subscripted variable
- Exploitation Method
- Performance Evaluation
- Conclusions

Intr. - Project Goal

- Our objective is to reduce memory access related power consumption
 - Power reduction is important
 - Proliferation of hand-held devices
Battery life is important
 - High power means costly packaging and cooling requirements, and lower reliability
 - Multimedia applications: data-dominated
 - Power consumption is largely contributed by data transfer and memory access operations

Intr. – Means to reach the Goal

- Different ways to reduce Power
 - System level – sleep mode, re-timing,...
 - Architecture level – multi-port v.s. single port
 - RTL block (registers) – I/O protocol (polling v.s. interrupt)..
 - Gate, transistors,...
- We choose to work on system level (high-level)
 - Better understanding of the system design, better control of data transfer and memory access
 - System designers use ad-hoc approaches to make decision starting at the specification

Intr. – Contributions

- A new method to exploit **multiple-index subscripted variable** (e.g. $x[i+j]$)
 - Previous works on register allocation restricted to **single-index subscripted variable** (e.g. $x[i]$)
- Extend the original concept of **self-temporal reuse**
 - Original concept of self-temporal reuse proposed by Monica S. Lam is generated by single-index subscripted variable

Multiple-Index Subscripted Variable

➤ Multiple-index subscripted variable

- Array reference with each dimension has one or more loop index variables
- Example: $x[i+j]$ has two loop indices i and j
- General formal

```
for (I1 = b1; I1 < (b1' + 1); I1 += t1)  
  for (I2 = b2; I2 < (b2' + 1); I2 += t2)  
    ...  
    for (IM = bM; IM < (bM' + 1); IM += tM)  
      ...x[ f( $\vec{I}$ ) ]+...
```

$$\vec{f}(\vec{I}) = \vec{I}H + \vec{c} = \begin{bmatrix} I_1 & I_2 & \dots & I_M \end{bmatrix} \begin{bmatrix} h_{11} & h_{12} & \dots & h_{1N} \\ h_{21} & h_{22} & \dots & h_{2N} \\ \dots & \dots & \dots & \dots \\ h_{M1} & h_{M2} & \dots & h_{MN} \end{bmatrix} + \begin{bmatrix} c_1 & c_2 & \dots & c_N \end{bmatrix}$$

or

$$\vec{f}(\vec{I}) = [f_1(\vec{I}) \quad f_2(\vec{I}) \quad \dots \quad f_N(\vec{I})]$$
$$= [I_1 h_{11} + I_2 h_{21} + \dots + I_M h_{M1} + c_1 \quad I_1 h_{12} + I_2 h_{22} + \dots + I_M h_{M2} + c_2 \quad \dots \quad I_1 h_{1N} + I_2 h_{2N} + \dots + I_M h_{MN} + c_N]$$



Multiple-Index Subscripted Variable

➤ Single-index subscripted variable

- Array reference with each dimension has only one loop index variables
- Example: $x[i]$ has only one loop index i

➤ Self-temporal reuse

- Occurs when a reference accesses the same datum at different loop iterations

- Example:

```
for (i = 0; i < 10; i++)  
  for (j = 0; j < 10; j++)  
    ... = x[i] ...
```

(a) single-index subscripted reference

```
for (i = 0; i < 10; i++)  
  for (j = 0; j < 10; j++)  
    ... = x[i+j] ...
```

(b) multiple-index subscripted reference



Multiple-Index Subscripted Variable

```
for (i = 0; i < 10; i++)  
  for (j = 0; j < 10; j++)  
    ... = x[i] ...
```

(a) single-index subscripted reference

```
for (i = 0; i < 10; i++)  
  for (j = 0; j < 10; j++)  
    ... = x[i+j] ...
```

(b) multiple-index subscripted reference

- $x[i]$ is a single-index subscripted variable
 - Its subscript is invariant to loop j
 - Its temporal reuse results from invariant loop j
 - $x[i]$ is referred to as **loop invariant reference**
 - $x[i]$ has 10 instances from $x[0]$ to $x[9]$
 - Every instance is accessed 10 times

Multiple-Index Subscripted Variable

```
for (i = 0; i < 10; i++)  
  for (j = 0; j < 10; j++)  
    ... = x[i] ...
```

(a) single-index subscripted reference

```
for (i = 0; i < 10; i++)  
  for (j = 0; j < 10; j++)  
    ... = x[i+j] ...
```

(b) multiple-index subscripted reference

- $x[i+j]$ is a multiple-index subscripted variable
 - Its subscript is a function of both i and j
 - Its temporal reuse results from both i and j
 - Its Instances are accessed with different number of times (e.g. $x[0]$ is accessed once while $x[9]$ is accessed 10 times)

Multiple-Index Subscripted Variable

- A general form of multiple-index subscripted variable $x[\vec{f}(\vec{I})]$ generates temporal reuse in a more complicated way than that of a single-index subscripted variable
- A method is needed to exploit temporal reuse generated by multiple-index subscripted variable

Exploitation Method

- Minimize memory access by exploiting the self-temporal reuse from multiple-index subscripted variables
 - Place the frequently access variable into a scalar variable to improve the likelihood of register allocation (**scalar replacement**)
 - Assumption: normal compilers using coloring-based register allocator will allocate scalar variable to a register
- The flow of the overall approach



Step 1 - Detection Process

- For a reference $x[\vec{f}(\vec{I})]$, self-temporal reuse exist when $\vec{f}(\vec{i}) = \vec{f}(\vec{j})$ and $\vec{i} \neq \vec{j}$
- \vec{i} and \vec{j} are two iterations of the loop nest

$$\vec{f}(\vec{I}) = \vec{I}H + \vec{c} = [I_1 \quad I_2 \quad \dots \quad I_M] \begin{bmatrix} h_{11} & h_{12} & \dots & h_{1N} \\ h_{21} & h_{22} & \dots & h_{2N} \\ \dots & \dots & \dots & \dots \\ h_{M1} & h_{M2} & \dots & h_{MN} \end{bmatrix} + [c_1 \quad c_2 \quad \dots \quad c_N]$$

or

$$\begin{aligned} \vec{f}(\vec{I}) &= [f_1(\vec{I}) \quad f_2(\vec{I}) \quad \dots \quad f_N(\vec{I})] \\ &= [I_1 h_{11} + I_2 h_{21} + \dots + I_M h_{M1} + c_1 \quad I_1 h_{12} + I_2 h_{22} + \dots + I_M h_{M2} + c_2 \quad \dots \quad I_1 h_{1N} + I_2 h_{2N} + \dots + I_M h_{MN} + c_N] \end{aligned}$$



Step 1 - Detection Process

- Find the distance between two iterations
 - \vec{d} is the distance vector computed by (1)

$$\because \vec{f}(\vec{i}) = \vec{f}(\vec{j}) \text{ and } \vec{i} \neq \vec{j}$$

$$\vec{i}H + \vec{c} = \vec{j}H + \vec{c}$$

$$(\vec{i} - \vec{j})H = \vec{0}, \text{ let } \vec{d} = \vec{i} - \vec{j}$$

$$\therefore \vec{d}H = \vec{0} \quad (1)$$

- Self-temporal exists when \vec{d} has a solution



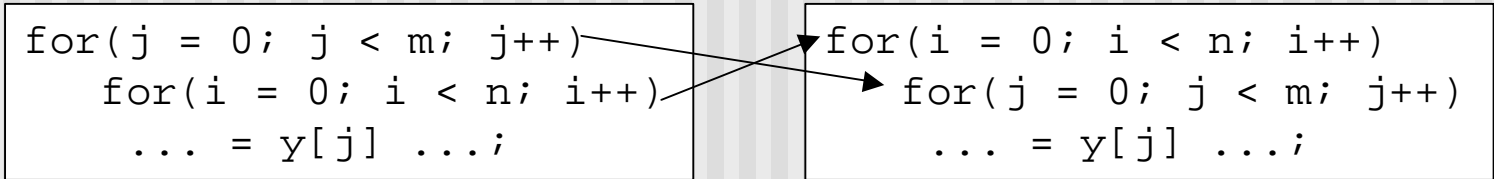
Step 2 - Loop Interchange

- **Loop interchange** is a loop transformation that permutes the loops inside a loop nest.

- Example:

```
for(j = 0; j < m; j++)  
  for(i = 0; i < n; i++)  
    ... = y[j] ...;
```

```
for(i = 0; i < n; i++)  
  for(j = 0; j < m; j++)  
    ... = y[j] ...;
```



- Loop interchange is performed to exploit loop invariant references
 - $y[j]$ is invariant to loop i , generating temporal reuse
 - Temporal reuse from loop invariant reference is exploitable only when the invariant loop is the inner most loop

Step 3 - Unroll-And-Jam

- A loop transformation that unrolls a loop body several times
 - Example: the outer loop is unrolled by v_1 times

```
for(i1=b1; i1<b1'; i1++)  
  for(i2=b2; i2<b2'; i2++)  
    body(i1, i2);
```

(a) Original loop



```
// unrolled loop  
for(i1=b1; i1<(b1'- b1'%v1); i1+=v1)  
{  
  for(i2=b2; i2<b2'; i2++)  
    body(i1, i2);  
  .....  
  for(i2=b2; i2<b2'; i2++)  
    body(i1+v1-1, i2);  
}  
// remainder loop  
for(i1=i1; i1<b1'; i1++)  
{  
  for(i2=b2; i2<b2'; i2++)  
    body(i1, i2);  
}
```

(b) Unroll outer loop by v_1 times

Step 3 - Unroll-And-Jam

- A loop transformation that unrolls a loop body several times
 - Example: Jamming multiple copies of inner loop

```
// unrolled loop
for(i1=b1; i1<(b1'- b1'%v1); i1+=v1)
{
    for(i2=b2; i2<b2'; i2++)
        body(i1,i2);
    .....
    for(i2=b2; i2<b2'; i2++)
        body(i1+v1-1,i2);
}
// remainder loop
for(i1=i1; i1<b1'; i1++)
{
    for(i2=b2; i2<b2'; i2++)
        body(i1,i2);
}
```

(b) Unroll outer loop by v_1 times



```
// unrolled loop after jamming
for(i1=b1; i1<(b1'- b1'%v1); i1+=v1)
{
    for(i2=b2; i2<b2'; i2++)
    {
        body(i1,i2);
        .....
        body(i1+v1-1,i2);
    }
}
// remainder loop (unchanged)
for(i1=i1; i1<b1'; i1++)
{
    for(i1=b1; i1<b1'; i1++)
        body(i1,i2);
}
```

(c) Jamming inner loops

Step 3 - Unroll-And-Jam

- Unroll-and-jam exposes temporal reuse generated by array reference
- Used in conjunction with scalar replacement to improve register allocation
- Find an optimal unrolling vector for a loop nest
 - Optimal unrolling vector reveals a significant amount of reduction in memory access and avoid register spilling

Step 3 - Unroll-And-Jam

- Find an optimal unrolling vector for a loop nest
 - Use a parameters estimation model
 - Parameters are:
 - Total number of reuse exposed by unroll-and-jam
 - Number of scalar variables for scalar replacement (less than the number of available registers)
 - Total number of memory access after the scalar replacement
 - Ratio of memory access after and before the optimization

Step 3 - Unroll-And-Jam

- Find an optimal unrolling vector for a loop nest
 - Iterative search of optimal unrolling vector
 - During the search, parameters estimation is performed for every possible unrolling vector
 - The unrolling vector leading to minimal memory access under the register restriction will be chosen

Step 3 - Unroll-And-Jam

- Find an optimal unrolling vector for a loop nest
 - Snapshot of the Iterative search

For every loop carrying legal temporal reuses, its unrolling factor v is initialized to the corresponding t_m . The minimal ratio R^{min} is initialized to 1. The optimum unrolling factor v^{opt} is initialized to t_m .

Assuming there are L loops carrying legal temporal reuses,

for $m=1$, to L

for $v_m = t_m$ to v_m^{max}

compute the number of reuses, Nr .

compute the number of registers, Nr .

if ($Nr <$ the number of registers available)

break;

endif.

compute the total number of memory accesses, Na .

compute the number of memory access ratio R .

if ($R < R^{opt}$ || (($R == R^{opt}$) && ($(v_1 \times v_2 \dots \times v_m \dots \times v_L) < (v_1^{opt} \times v_2^{opt} \dots \times v_m^{opt} \dots \times v_L^{opt})$)))

$R^{opt} = R$.

Optimal unrolling vector is assigned with the current unrolling vector.

endif

endfor

endfor



Step 4 – Scalar Replacement

- Scalar replacement replaces subscripted variables by temporary scalar variables to effect reuse
- Apply in round robin fashion

- Example:

```
for(i=0; i<N; i++)  
    ... = x[i] + x[i+1];
```

Unroll loop by 2

```
for(i=0; i<(N-N%2); i+=2)  
{  
    ... = x[i] + x[i+1];  
    ... = x[i+1] + x[i+2];  
}  
For(i=i; i<N; i++)  
    ... = x[i] + x[i+1];
```

Scalar a, b and c
used in round
robin fashion

```
b = x[0]; c = x[1];  
for(i=0; i<(N-N%2); i+=2)  
{  
    a=b;  
    b=c;  
    c=x[i+2];  
    ... = a + b;  
    ... = b + c;  
}  
For(i=i; i<N; i++)  
    ... = x[i] + x[i+1];
```



Performance Evaluation

- The method is applied to five benchmarks taken from the High-level Synthesis Design Repository (HLSynth95/memory)
- The optimized benchmarks are executed on SimpleScalar

TABLE I
BENCHMARK FROM HLSYNTH95/MEMORY

Benchmark	Description
Compression	An image compression scheme
Laplace	A Laplace algorithm to perform edge enhancement
LowPass	A low-pass filter to an image
SOR	A Successive Over-Relaxation (SOR) algorithm
Wavelet	The Daubechies 4-Coefficient Wavelet filter

Performance Evaluation

- Performance of the method is measured in terms of:
 - Accuracy of the estimation model compared with practical results from simulation
 - Percentage of memory access reduction

Performance Evaluation

➤ Accuracy test

- Example: benchmark program – Compression
- R is the ratio of memory access after and before the optimization
- The estimated R is close to the practical R from simulation

TABLE II
COMPARISON OF ACCESS RATIOS FOR COMPRESSION UNDER
10-REGISTER RESTRICTION

[v1 v2]	Estimated R	Practical R
[2 2]	0.6370	0.6520
[2 3]	0.6040	0.6201
[3 2]	0.6040	0.6201

Performance Evaluation

- Percentage reduction in memory access
 - Results show that significant reductions to the benchmarks are achieved by our method

TABLE III
PERCENTAGE OF MEMORY ACCESS REDUCTION

Benchmark	Optimal Vector	No. of Reg. Used	Memory Access Reduction (%)
Compression	[2 3]	8	0.3799
Laplace	[1 3]	9	0.3919
Lowpass	[1 3]	9	0.3919
SOR	[2 3]	10	0.1347
Wavelet	6	10	0.4167



Conclusions

- A method for reducing the number of memory access by transform source code of data intensive program is proposed
- Advantages:
 - Exploration space is broadened by considering multiple-index subscripted variables comparing to previous works
 - Measurements designed in our approach give an accurate estimation of the practical effect.
 - Significant improvements achieved in memory access

Q & A

Thank you