[Chap.5-1] Optimizing Program Performance

Young Ik Eom (yieom@skku.edu, 031-290-7120)
Distributing Computing Laboratory
Sungkyunkwan University
http://dclab.skku.ac.kr

Contents

- Introduction
- Optimizing compilers
- Expressing program performance
- Benchmark example
- Loop optimization
- Reducing procedure calls
- Reducing memory references
- Understanding modern processors
- Loop unrolling
- Enhancing parallelism
- …

Introduction

- **Program code**
  - Should work correctly
  - Should be clear and concise
    - Readability, understandability, maintainability
  - Should run efficiently
    - Performance

Introduction

- **Efficient program code**
  - Must select an appropriate set of data structures and algorithms
  - Must have source code that the compiler can effectively optimize to turn into efficient executable code
  - Divide a task into portions that can be run in parallel
    - For utilizing multiple processors and multiple cores
  - Tradeoff between
    how easy a program is to implement and maintain,
    and how fast it runs

Introduction

- **Efficient program code**
  - Compiler tries to generate efficient code
    - Using several optimization techniques
  - But, compilers can be thwarted by **optimization blockers**
    - Potential **memory aliasing**, **procedure side-effects**
  - So, programmers must assist the compiler
    by writing code that can be optimized readily

Introduction

■ Efficient program code

- Some techniques for improving code performance
  - ① Eliminate unnecessary work **[target machine independent]**
    - ✓ Such as unnecessary function calls, conditional tests, and memory references
  - ② Exploit the capability of processors **[target machine dependent]**
    - ✓ Modern processor architectures and timing information of each operation
      - Parallel processing, out-of-order execution, etc…

Introduction

- **Code optimization**
  - Not straightforward
  - Needs a fair amount of trial-and-error experimentation

  - Good strategy
    - Inner loop inspection
    - Parallelism detection by identifying critical paths
    - Etc

Optimizing Compilers

- **Optimization level of gcc**
  - Command-line option **`-Og`**
    - Applies basic set of optimizations
  - Command-line option **`–O1`** and higher (**`-O2`** and **`–O3`**)
    - Applies more extensive optimizations
    - Further improves program performance
    - May expand the program size
    - May make the program more difficult to debug

We will be mostly consider code compiled with **`–O1`** in this chapter

We can write C code that, when compiled just with **`–O1`**,
vastly outperforms a more naive version
compiled with the highest possible optimization levels

Optimizing Compilers

- **Safe optimizations**
  - The optimized code should have the exact same behavior as the un-optimized version
    - It is necessary to check an optimization is safe or not
  - Typical optimization blockers
    - Memory aliasing
    - Procedure side-effects

Optimizing Compilers

Safe optimizations

- Optimization blockers: **Memory aliasing**

- Example1) **twiddle1** and **twiddle2**

```
void twiddle1(int *xp, int *yp)
{
    *xp = *xp + *yp;
    *xp = *xp + *yp;
}
```

6 memory references

```
void twiddle2(int *xp, int *yp)
{
    *xp = *xp + 2 * *yp;
}
```

More efficient
3 memory references

Optimizing Compilers

■ Safe optimizations

- Optimization blockers: **Memory aliasing**

- Example1) **twiddle1** and **twiddle2**

```
void twiddle1(int *xp, int *yp)
{
    *xp = *xp + *yp;
    *xp = *xp + *yp;
}
```

6 memory references

```
void twiddle2(int *xp, int *yp)
{
    *xp = *xp + 2 * *yp;
}
```

More efficient
3 memory references

Same behavior ?

Optimizing Compilers

- **Safe optimizations**
  - Optimization blockers: **Memory aliasing**

  - Example1) **twiddle1** and **twiddle2**

```
void twiddle1(int *xp, int *yp)
{
    *xp = *xp + *yp;
    *xp = *xp + *yp;
}
```

6 memory references

```
void twiddle2(int *xp, int *yp)
{
    *xp = *xp + 2 * *yp;
}
```

More efficient
3 memory references

Same behavior ?

Optimization possible ?

Optimizing Compilers

Safe optimizations

- Example1)
  - When **xp** and **yp** are equal (**memory aliasing**)
    - ✓ In **twiddle1**, **xp** is increased by a factor of
    - ✓ In **twiddle2**, **xp** is increased by a factor of

Two pointers points to the same memory location

```
void twiddle1(int *xp, int *yp)
{
    *xp = *xp + *yp;
    *xp = *xp + *yp;
}
```

```
void twiddle2(int *xp, int *yp)
{
    *xp = *xp + 2 * *yp;
}
```

Not same when xp == yp

Optimization not possible !

Optimizing Compilers

Safe optimizations

- Optimization blockers: **Memory aliasing**

- Example2)

```
X = 1000;
y = 3000;
*q = y;
*p = x;
t = *q;
```

  - The value of **t** depends on
    whether or not the pointers **p** and **q** are aliased

- Memory aliasing is one of the major optimization blockers
  - Limits the set of possible optimizations

Optimizing Compilers

- **Safe optimizations**
  - Optimization blockers: **Procedure side effects**
  - Example)

```
int f();

int f1() {
  return f() + f() + f() + f();
}
```

calls f() 4 times

```
int f();

int f2()
{
  return 4*f();
}
```

calls f() 1 time

Optimizing Compilers

- **Safe optimizations**
  - Optimization blockers: **Procedure side effects**
  - Example)

calls f() 4 times

```
int f();

int f1() {
  return f() + f() + f() + f();
}
```

Optimization possible ?

```
int f();

int f2()
{
  return 4*f();
}
```

calls f() 1 time

Optimizing Compilers

- **Safe optimizations**
  - Optimization blockers: **Procedure side effects**
  - Example)

```
int c = 0;

int f() {
  return c++;
}
```

```
int f();

int f1() {
  return f() + f() + f() + f();
}
```

returns

```
int f();

int f2()
{
  return 4*f();
}
```

returns

Optimization not possible !

Expressing Pgm Performance

CPE (Cycles Per Element)

- Convenient way to express program performance that operates on vectors or lists (loop)
- Effective number of cycles consumed for processing 1 element

- Cycles
  - Processor speed: # of cycles per second
    - ✓ Ex) 4GHz → 4.0×10^9 cycles per second
  - 1 micro-operation per cycle
  - A machine instruction requires several cycles for execution

Expressing Pgm Performance

- **CPE (Cycles Per Element)**
  - Total time consumed for a procedure with loop (**T**)
    - **T = a·n + b** (in number of cycles)
      where **n** is the number of elements in the list
    - **a → CPE**
    - **b → Overhead**

↔ Cycles per iteration

Expressing Pgm Performance

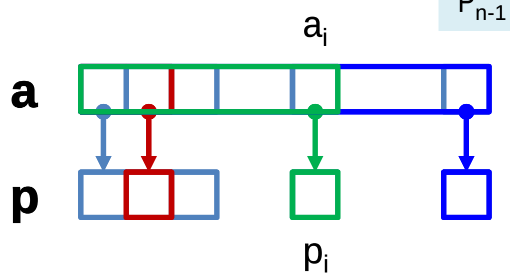
Example) Computing prefix sum

- For a vector **A** = $<a_0, a_1, \ldots, a_{n-1}>$, the prefix sum **P** = $<p_0, p_1, \ldots, p_{n-1}>$ is defined as

  $p_0 = a_0$

  $p_i = p_{i-1} + a_i,\ 1 \le i < n$

$p_0 = a_0$
$p_1 = a_0+a_1$
$\cdots$
$p_i = a_0+a_1+\cdots+a_i$
$\cdots$
$P_{n-1} = a_0+a_1+\cdots+a_{n-1}$



- Function **psum1()**
  - Computes 1 element of the result vector per iteration
- Function **psum2()**
  - Loop unrolling
  - Computes 2 elements per iteration

Expressing Pgm Performance

- Example) Computing prefix sum

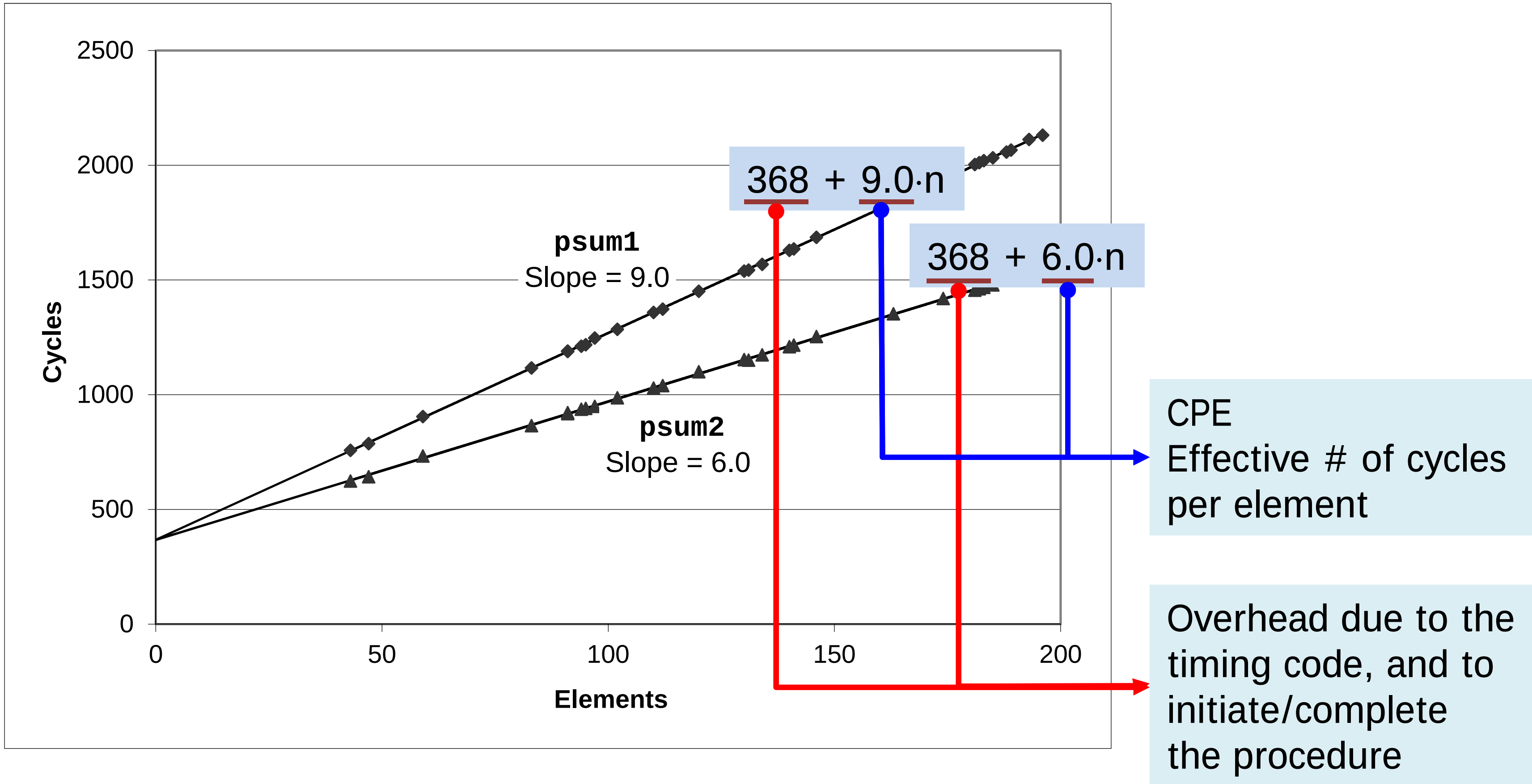
```
void psum1(float a[], float p[], long n)
{
   long i;
   p[0] = a[0];
   for (i=1; i<n; i++)
      p[i] = p[i-1] + a[i];
}
```

```
void psum2(float a[], float p[], long n)
{
   long i;
   p[0] = a[0];
   for (i=1; i<n-1; i+=2) {
      float mv = p[i-1] + a[i];
      p[i] = mv;
      p[i+1] = mv + a[i+1];
   }
   if (i < n)
      p[i] = p[i-1] + a[i];
}
```

Expressing Pgm Performance

Example) Computing prefix sum

- Performance of prefix sum functions

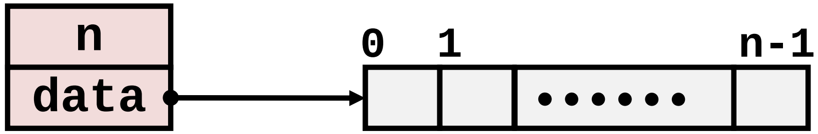


Benchmark Example

- **Measuring CPE performance on an example**
  - Reference machine
    - Intel Core i7 Haswell
  - Example program
    - Vector operation
  - CPE measurements
    - Integer, single-precision FP data, double-precision FP data
    - Addition, multiplication

Benchmark Example

- Vector abstract data type




```
/* data structure for vectors */
typedef struct{
   long n;
   data_t *data;
} v_r, *v_p;

/* data_t can be of any type (int, long, float, double, ...) */
```

Benchmark Example

- **Basic functions**

```
/* retrieve vector element and store it at dest */
int get_v_element(v_p v, long idx, data_t *dest)
{
   if (idx < 0 || idx >= v->n)
      return 0;
   *dest = v->data[idx];
   return 1;
}
```

```
/* return length of vector */
long v_length(v_p v)
{
   return v->n;
}
```

Benchmark Example

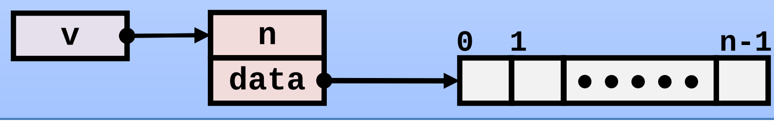
■ Target function (version-1)

```
void combine1(v_p v, data_t *dest)
{
    long i;

    *dest = IDENT;
    for (i = 0; i < v_length(v); i++) {
        data_t val;
        get_v_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

```
#define IDENT 0
#define OP +
```

```
#define IDENT 1
#define OP *
```



Function	Method	Integer		FP	
		+	*	+	*
combine1	Abstract unoptimized	22.68	20.02	19.98	20.18
combine1	Abstract –O1	10.12	10.12	10.17	11.14

Loop Optimization

Target function (version-2) [eliminating loop inefficiencies]

```
void combine2(v_p v, data_t *dest)
{
    long i;
    long length = v_length(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        data_t val;
        get_v_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Code motion
Identifying a computation that is performed multiple times, but the result will not change

Consider the machine code for function call (Chap.3)

Eliminated the call to **v_length()** from the loop and placed it at the beginning of the function

Function	Method	Integer		FP	
		+	*	+	*
combine1	Abstract –O1	10.12	10.12	10.17	11.14
combine2	Move `v_length()`	7.02	9.03	9.02	11.03

Loop Optimization

Another example) code motion

```
void lower1(char *s)
{
   int i;
   for (i = 0; i < strlen(s); i++)
      if (s[i] >= 'A' && s[i] <= 'Z')
         s[i] -= ('A' - 'a');
}
```

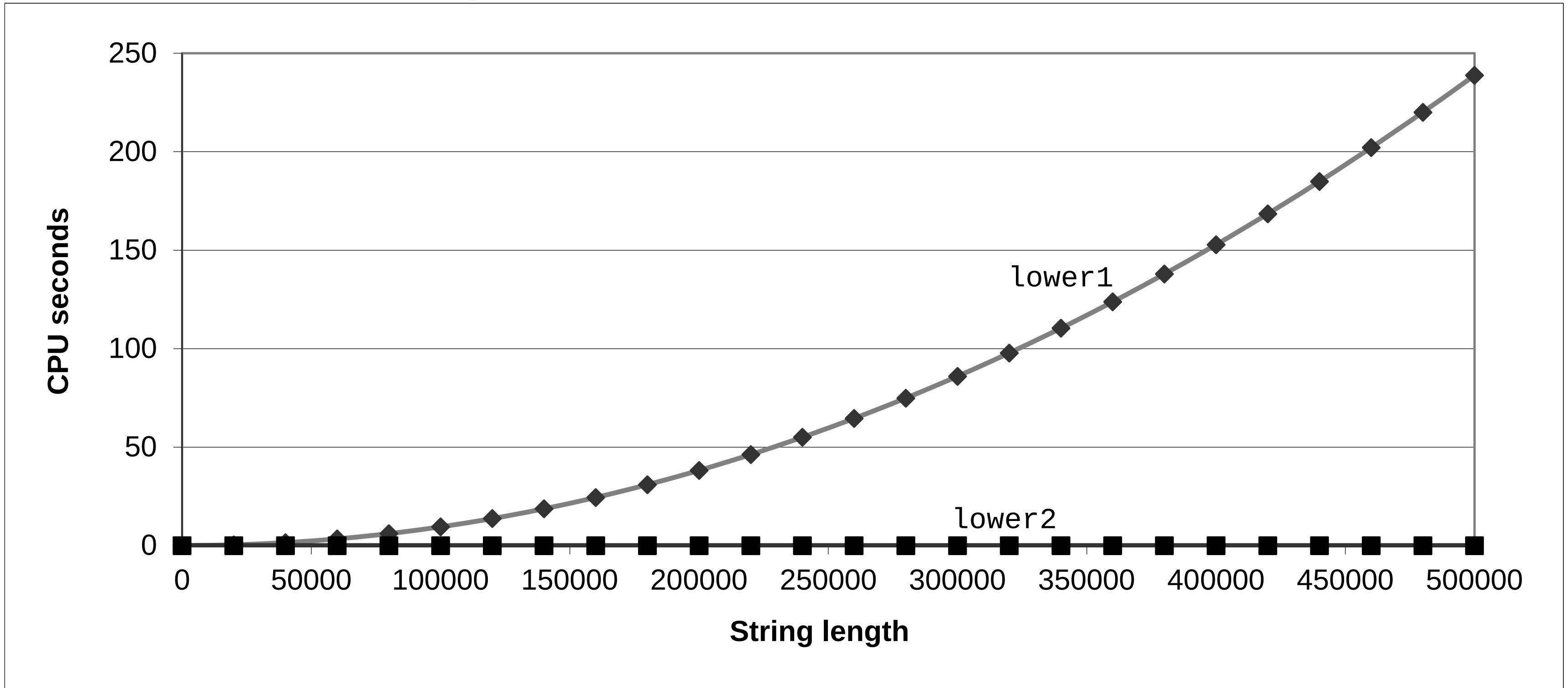
```
void lower2(char *s)
{
   int i;
   int len = strlen(s);

   for (i = 0; i < len; i++)
      if (s[i] >= 'A' && s[i] <= 'Z')
         s[i] -= ('A' - 'a');
}
```

```
/* My version of strlen */
size_t strlen(const char *s)
{
   size_t length = 0;
   while (*s != '\0') {
      s++;
      length++;
   }
   return length;
}
```

Loop Optimization

■ Another example) code motion (actual measurements)



Function	String length						
	16,384	32,768	65,536	131,072	262,144	524,288	1,048,576
lower1	0.26	1.03	4.10	16.41	65.62	262.48	1,049.89
lower2	0.0000	0.0001	0.0001	0.0003	0.0003	0.0005	0.0020

Loop Optimization

Another example) code motion

- Can the compiler can do this type of code motion automatically?
  - Requires very sophisticated analysis
    - ✓ Since **strlen** checks the elements of the string and these values are changing as **lower1** proceeds
    - ✓ The compiler should be able to confirm that none of the elements are being set from non-zero to zero
  - So, programmers must do such transformations themselves

Reducing Procedure Calls

Target function (version-3) [reducing procedure calls]

```
void combine3(v_p v, data_t *dest)
{
    long i;
    long length = v_length(v);
    data_t *data = get_v_start(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

Rather than making a call to get each vector element, it accesses the array directly

Eliminated the call to **get_v_element()** from the loop

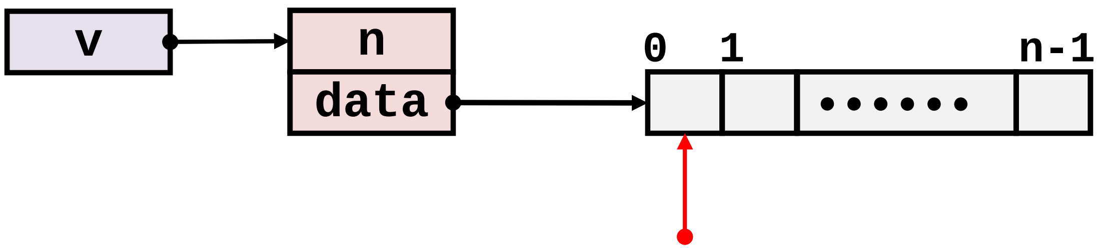
Function	Method	Integer		FP	
		+	*	+	*
combine2	Move **v_length()**	7.02	9.03	9.02	11.03
combine3	Direct data access	7.17	9.02	9.02	11.03

No apparent performance improvement !!!

Reducing Procedure Calls

■ 1 more basic functions (for version-3)

```
/* returns the start address of the data array */
data_t *get_v_start(v_p v)
{
   return v->data;
}
```



Reducing M. References

Target function (version-3)

- x86-64 code for data type **double** and with multiplication

```
[Inner loop of combine3()]
   data_t = double, OP = *
   dest in %rbx, data+i in %rdx, data+length in %rax
.L17:                               loop:
   vmovsd (%rbx),%xmm0                 Read product from dest
   vmulsd (%rdx),%xmm0,%xmm0           Multiply product by data[i]
   vmovsd %xmm0,(%rbx)                 Store product at dest
   addq $8,%rdx                        Increment data+i
   cmpq %rax,%rdx                      Compare to data+length
   jne .L17                            If !=, goto loop
```

- Wasteful reading and writing from and to the **dest** (**%rbx**)
- Can reduce the memory operations by using a register for **dest**

Reducing M. References

■ Target function (version-4) [reducing memory references]

```
void combine4(v_p v, data_t *dest)
{
   long i;
   long length = v_length(v);
   data_t *data = get_v_start(v);
   data_t acc = IDENT;

   for (i = 0; i < length; i++) {
      acc = acc OP data[i];
   }
   *dest = acc;
}
```

Eliminated needless reading and writing of memory (another var **acc**)

Function	Method	Integer		FP	
		+	*	+	*
combine3	Direct data access	7.17	9.02	9.02	11.03
combine4	Accumulate in local var acc	1.27	3.01	3.01	5.01

Reducing M. References

Target function (version-4)

- x86-64 code for data type **double** and with multiplication

```
[Inner loop of combine4()]
   data_t = double, OP = *
   acc in %xmm0, data+i in %rdx, data+length in %rax

.L25:                                       loop:
   vmulsd (%rdx),%xmm0,%xmm0                   Multiply acc by data[i]
   addq $8,%rdx                                Increment data+i
   cmpq %rax,%rdx                              Compare to data+length
   jne .L25                                    If !=, goto loop
```

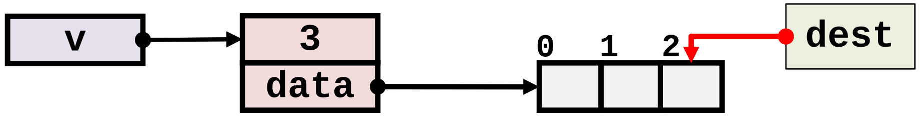
- Reduced the memory operations by using a register for **acc**
- All of the times are improved by factors of 2.2× ~ 5.7×

Reducing M. References

Note) Memory aliasing in combine3()

```
combine3(v, get_v_start(v) + 2);
combine4(v, get_v_start(v) + 2);
```

Aliasing between the last element of the 3 integer element vector and the destination



- Traces of the array values (case integer and multiplication)

Function	Initial	Before loop	i = 0	i = 1	i = 2	Final
combine3	[2, 3, 5]	[2, 3, 1]	[2, 3, 2]	[2, 3, 6]	[2, 3, 36]	[2, 3, 36]
combine4	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 5]	[2, 3, 30]

- Can the compiler transform the **combine3** code to accumulate the value in register as it does with the **combine4** code? No!

Benchmark Example Summary

■ Summary

아키텍쳐에 독립적

종속적

Function	Method	Integer		FP	
		+	*	+	*
combine1	Abstract unoptimized	22.68	20.02	19.98	20.18
combine1	Abstract –O1	10.12	10.12	10.17	11.14
combine2	Move **`v_length()`**	7.02	9.03	9.02	11.03
combine3	Direct data access	7.17	9.02	9.02	11.03
combine4	Accumulate in local var acc	1.27	3.01	3.01	5.01
…					

Summary