[Chap.5-3] Optimizing Program Performance

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

Contents

- …
- **Some limiting factors**
- **Understanding M performance**
- **Program profiling**

Some Limiting Factors

- **Register spilling**
  - When we have a degree of parallelism that exceeds the number of available registers, the compiler will resort to register spilling
    - Allocating some of the temporary values on memory (stack)
    - The performance can drop significantly

Function	Method	Integer		FP	
		+	*	+	*
combine6					
	10×10 unrolling	0.55	1.00	1.01	0.52
	20×20 unrolling	0.83	1.03	1.02	0.68
Throughput bound		0.50	1.00	1.00	0.50

Some Limiting Factors

- **Register spilling**
  - Example) Code for accumulating **acc0** in **combine6**
    - Case of 10×10 unrolling and 20×20 unrolling

```
[Inner loop of combine6() with 10×10 unrolling]
   Updating of accumulator acc0 in 10×10 unrolling

   vmulsd (%rdx),%xmm0,%xmm0     acc0 on %xmm0
```

```
[Inner loop of combine6() with 20×20 unrolling]
   Updating of accumulator acc0 in 20×20 unrolling

   vmovsd 40(%rsp),%xmm0         acc0 on 40(%rsp) in stack
   vmulsd (%rdx),%xmm0,%xmm0
   vmovsd %xmm0,40(%rsp)
```

- In 20×20 unrolling, acc0 is allocated on the stack
- The advantages of multiple accumulators are likely to be lost

Some Limiting Factors

- **Branch prediction and mis-prediction penalties**
  - A conditional branch can incur a significant mis-prediction penalty, when the branch prediction logic does not correctly predict
    - Whether or not a branch will be taken
    - The target address

Some Limiting Factors

- **Branch prediction: challenges**
  - Example-1)

```
80489f3:  movq   $0x1,%rcx
80489f6:  xorq   %rdx,%rdx
80489f9:  cmpq   %rsi,%rdx
80489fc:  jnl    0x8048a25
80489fe:  movq   %rax,%rsi
8048a02:  imulq  (%rax,%rdx,4),%rcx
```

Executing

How to continue?

```
8048a25:  cmpq   %rdi,%rdx
8048a28:  jl     0x8048a20
8048a2a:  movq   0xc(%rbp),%rax
8048a2d:  leaq   0xffffffe8(%rbp),%rsi
```

Some Limiting Factors

- **Branch prediction: outcomes**
  - Example-1)

```
80489f3:  movq   $0x1,%rcx
80489f6:  xorq   %rdx,%rdx
80489f9:  cmpq   %rsi,%rdx
80489fc:  jnl    0x8048a25
80489fe:  movq   %rax,%rsi
8048a02:  imulq  (%rax,%rdx,4),%rcx
```

Branch not-taken

Branch taken

```
8048a25:  cmpq   %rdi,%rdx
8048a28:  jl     0x8048a20
8048a2a:  movq   0xc(%rbp),%rax
8048a2d:  leaq   0xffffffe8(%rbp),%rsi
```

Some Limiting Factors

Branch prediction through loop

- Example-2)

Assume vector length = 100

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 0 — Executed

Predict taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 1 — Executed

Predict taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 2 — Executed

Predict taken (OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```
i = 3 — Fetched

Some Limiting Factors

Branch prediction through loop

- Example-2)

Assume
vector length = 100

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 98

Predict taken
(OK)

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 99

Predict taken
(OOPs)

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 100

Read invalid
location

Executed

```
401029:   vmulsd  (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
```
i = 101

Fetched

Some Limiting Factors

Branch mis-prediction invalidation

- Example-2)

Assume
vector length = 100

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 98

Predict taken
(OK)

```
401029:  vmulsd (%rdx),%xmm0,%xmm0
40102d:  add    $0x8,%rdx
401031:  cmp    %rax,%rdx
401034:  jne    401029
```

i = 99

Predict taken
(OOPs)

Invalidate

~~401029: vmulsd (%rdx),%xmm0,%xmm0~~
~~40102d: add $0x8,%rdx~~
~~401031: cmp %rax,%rdx~~
~~401034: jne 401029~~

i = 100

Executed

~~401029: vmulsd (%rdx),%xmm0,%xmm0~~
~~40102d: add $0x8,%rdx~~
~~401031: cmp %rax,%rdx~~
~~401034: jne 401029~~

i = 101

Fetched

Some Limiting Factors

- **Branch prediction and mis-prediction penalties**
  - Effect of branch prediction

```
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;
}
```

combine4

```
void combine4b(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++)
      if (i >= 0 && i < v->len)
         acc = acc OP v->data[i];
   *dest = acc;
}
```

combine4b

Function	Method	Integer		FP	
		+	*	+	*
combine4	No bounds checking	1.27	3.01	3.01	5.01
combine4b	Bounds checking	2.02	3.01	3.01	5.01

Some Limiting Factors

- **Branch prediction and mis-prediction penalties**
  - Effect of branch prediction
  - So, …
    - Do not be overly concerned about predictable branches
      - ✓ With loops, only misses when hits loop end
        (regular patterns, highly predictable)
    - Write code suitable for implementation with conditional moves
      - ✓ For inherently unpredictable cases

Some Limiting Factors

- **Branch prediction and mis-prediction penalties**
  - Utilizing conditional data transfers
    - No strict rules
    - Coding in an functional style rather than imperative style

Some Limiting Factors

- **Branch prediction and mis-prediction penalties**
  - Utilizing conditional data transfers

```
void minmax1(long a[], long b[], long n)
{
   long i;
   for (i = 0; i < n; i++)
      if (a[i] > b[i]) {
         long t = a[i];
         a[i] = b[i];
         b[i] = t;
      }
}
```

Imperative

```
void minmax2(long a[], long b[], long n)
{
   long i;
   for (i = 0; i < n; i++) {
      long min = a[i] < b[i] ? a[i] : b[i];
      long max = a[i] < b[i] ? b[i] : a[i];
      a[i] = min;
      b[i] = max;
   }
}
```

Functional

Getting High Performance

- **Basic strategies for optimizing program performance**
  - High-level design
    - Appropriate algorithms and data structures
    - Time complexity and space complexity
  - Basic coding principles
    - Watch out for optimization blockers and help compiler to produce efficient code
      - ✓ Move computations out of loops when possible
      - ✓ Eliminate excessive function calls
      - ✓ Eliminate unnecessary memory references
    - Look carefully at inner loops

Getting High Performance

- **Basic strategies for optimizing program performance**
  - Low-level optimizations
    - Structure code to take advantage of the hardware capabilities
      - ✓ Unroll loops
      - ✓ Find ways to increase instruction-level parallelism
        - Multiple accumulators and re-associations
      - ✓ Avoid unpredictable branches
      - ✓ Try to rewrite conditional operations in a functional style to enable compilation via conditional data transfers
    - Make code cache friendly (will be covered later)

Program Profiling

Code profiler

- Tools to catch on where to focus our optimization efforts
  - Used in developing large-scale programs

Program profiling

- Running a version of a program in which instrumentation code has been incorporated to determine how much time the different parts of the program require
- **gprof**
  - Profiling utility in Unix
  - 2 forms of outputs
    - ✓ CPU time spent by each function in the program (flat profile)
    - ✓ Count on how many times each function gets called (call graph)

Program Profiling

Program profiling

- 3 steps in profiling with **gprof**
  - Compiling and linking for profiling
    - ✓ **gcc** option **–pg**

```
linux> gcc -Og -pg prog.c -o prog
```

  - Execution of the program as usual
    - ✓ Runs slightly slowly than normal
    - ✓ Generates a file **gmon.out**

```
linux> ./prog file.text
```

  - Invoking **gprof** to analyze the data in **gmon.out**

```
linux> gprof prog
```

Program Profiling

Program profiling

- Profile report
  - Times spent executing each function

Flat profile

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
97.58	203.66	203.66	1	203.66	203.66	sort_words
2.32	208.50	4.85	965027	0.00	0.00	find_ele_rec
0.14	208.81	0.30	12511031	0.00	0.00	Strlen

  - Calling history of the functions (for **find_ele_rec**)

Call graph

index	%time	self	children	called	name
				158655725	find_ele_rec [5]
		4.85	0.10	965027/965027	insert_string [4]
[5]	2.4	4.85	0.10	965027+158655725	find_ele_rec [5]
		0.08	0.01	363039/363039	save_string [8]
		0.00	0.01	363039/363039	new_ele [12]
				158655725	find_ele_rec [5]

Program Profiling

Program profiling

- Some properties of **gprof**
  - The timing information is not precise
    - ✓ Simply based on interval counting scheme
    - ✓ Does not reflect the interventions of kernel (such as interrupts)
  - The calling information is quite reliable
  - The timings of library functions are not shown
    - ✓ Incorporated into the times of the calling function

  - Ref) https://sourceware.org/binutils/docs/gprof/index.html

Program Profiling

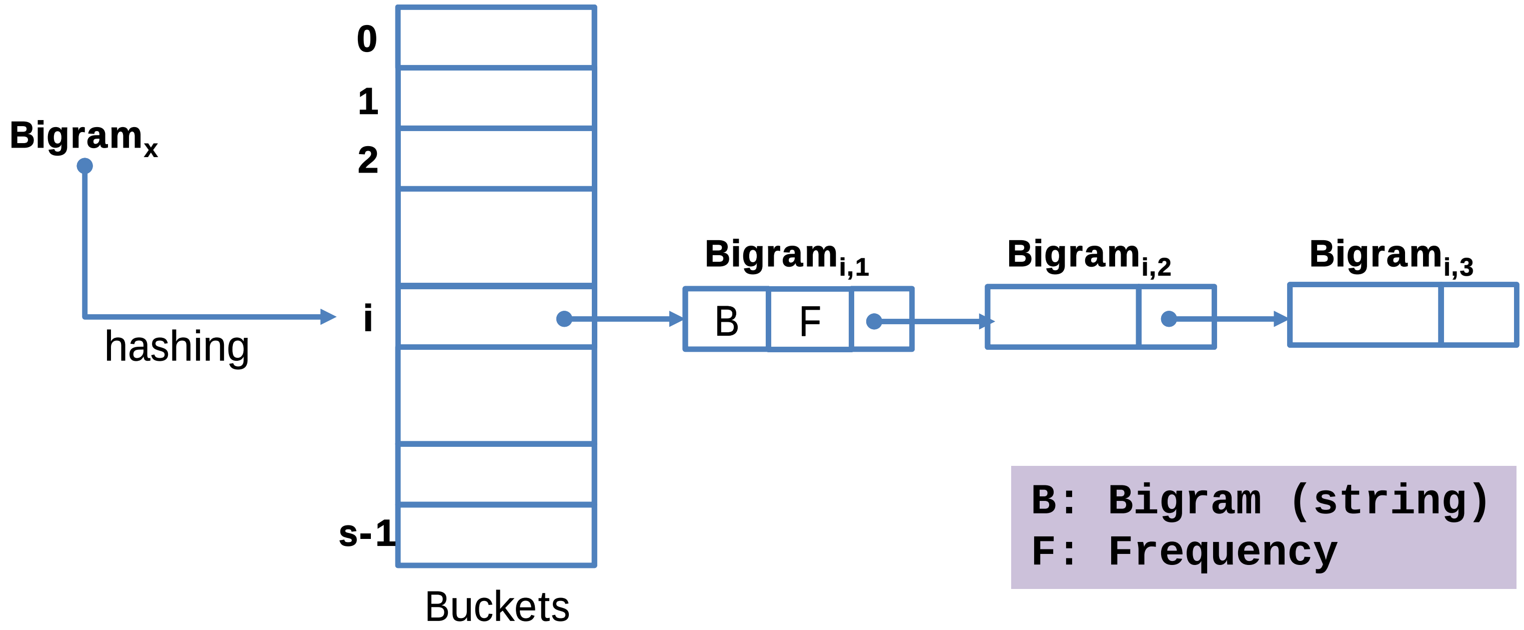
Using a profiler

- Target application
  - A program that analyzes the **n-gram** statistics of a text document, where **n-gram** means a sequence of **n** words occurring in a document (bigram for the case n = 2)
  - Uses works of William Shakespeare as input data
    - ✓ 965,028 words; 363,039 unique bigrams
  - Program components
    - ✓ Conversion of each word to lowercase
    - ✓ Hashing to **s** buckets
    - ✓ Scanning each hash bucket (linked list)
    - ✓ Sorting all elements according to the frequencies

Program Profiling

Using a profiler

- Target application

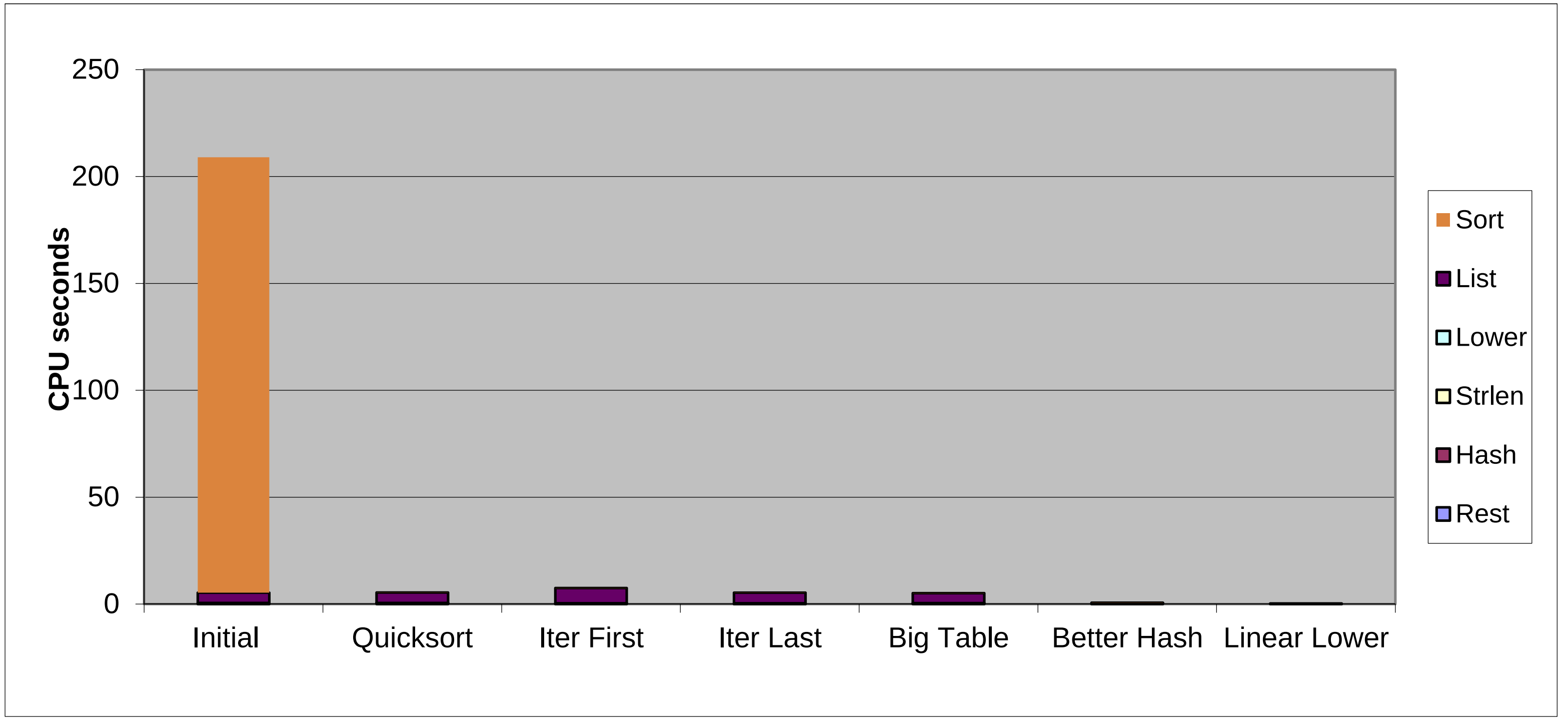


Program Profiling

- **Using a profiler**
  - Target application
    - Timing categories
      - ✓ Sort (sorting n-grams by frequency)
      - ✓ List (scanning the linked list, inserting new elements, if necessary)
      - ✓ Lower (converting strings to lowercase)
      - ✓ Strlen (computing string lengths)
      - ✓ Hash (computing hash functions)
      - ✓ Rest (sum of all other functions)
    - 6 versions for enhancing the target applications
      - ✓ …

Program Profiling

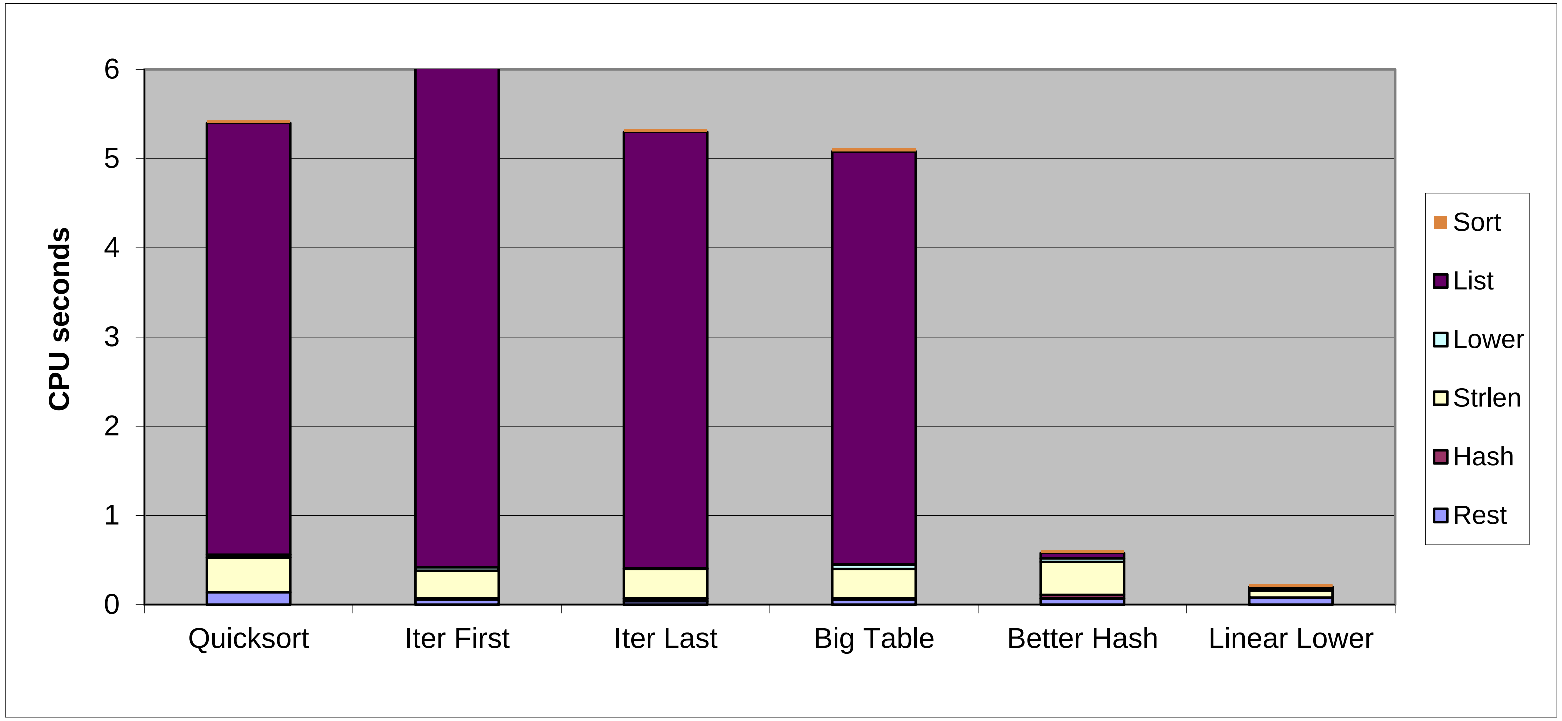
- **Using a profiler**
  - Profile results



Program Profiling

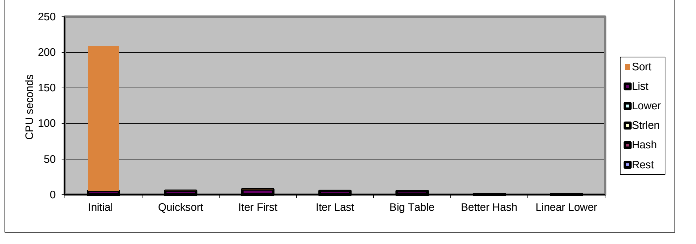
Using a profiler

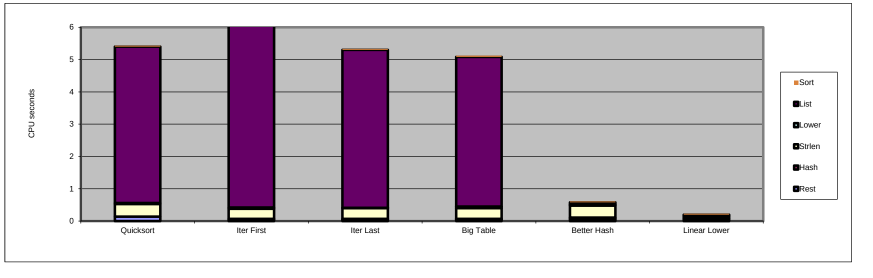
- Profile results



Program Profiling

- **Using a profiler**
  - 1st version of the n-gram analyzer
    - Insertion sort, **lower1()**, summing ASCII code values mod s, recursive list scanning
    - 3.5 minutes
      - ✓ Most of the time is spent for sorting
  - 2nd version
    - Quick sort
    - 5.4 seconds
      - ✓ Now, list scanning becomes the bottleneck

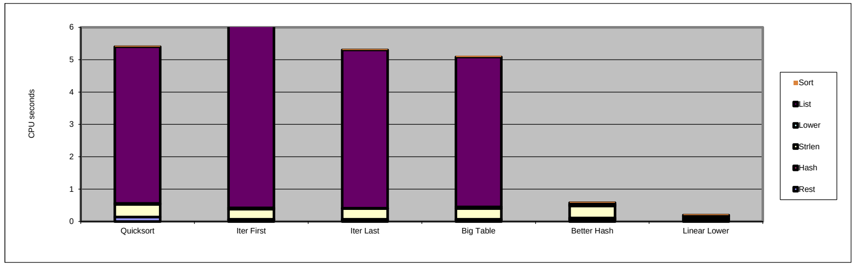




Program Profiling

■ Using a profiler

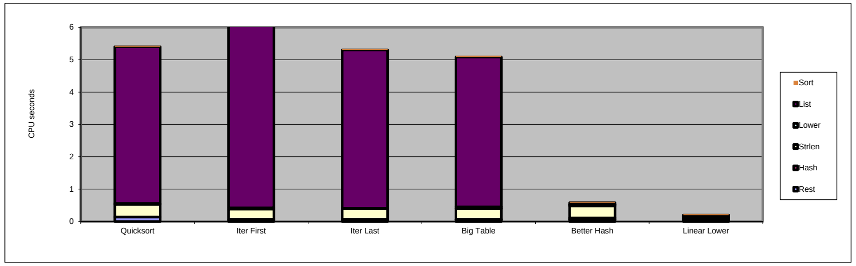
- 3^{rd} version
  - Iterative list scanning, inserting at the front of the list
  - 7.5 seconds
    - ✓ Still… the bottleneck is on the list scanning
- 4^{th} version
  - Iterative list scanning, inserting at the end of the list
  - 5.3 seconds
    - ✓ And still… the bottleneck is on the list scanning



Program Profiling

Using a profiler

- 5^{th} version
  - Increased # of buckets (199,999)
  - 5.1 seconds
    - ✓ Also… the bottleneck is on the list scanning
- 6^{th} version
  - Better hash function
  - 0.6 seconds
    - ✓ Now, we should focus on the **Strlen** function which calls **lower1()**
- 7^{th} version
  - Using **lower2()**
  - 0.2 seconds



Program Profiling

Using a profiler

- Summary
  - Code profiling helps to drop the time for the **n-gram** analyzer from nearly 3.5 minutes down to under 1 second
  - The profiler
    - ✓ Helps developers to focus their attention on the most time-consuming parts of the program
    - ✓ Provides useful information about the procedure call structure

Program Profiling

Amdahl's law

- Problem statement
  - **T** total time required for an application
  - **p** fraction of the total that can be sped up ($0 \le \mathbf{p} \le 1$)
  - **k** speedup factor
  - $\mathbf{T_k = ?}$ (overall execution time)
- Resulting performance
  - $\mathbf{T_k = (1\text{-}p)\cdot T + p\cdot T/k}$
    - ✓ Portion which can be sped up runs **k** times faster
    - ✓ Portion which cannot be sped up stays the same
  - Maximum possible speedup
    - ✓ $\mathbf{k = \infty}$
    - ✓ $\mathbf{T_\infty = (1\text{-}p)\cdot T}$

Program Profiling

Amdahl's law

- Example)
  - Overall problem
    - ✓ **T** = 10 Total time required
    - ✓ **p** = 0.9 Fraction of total which can be sped up
    - ✓ **k** = 9 Speedup factor
  - Resulting Performance
    - ✓ $T_9 = 0.1 * 10 + 0.9 * 10/9 = 1.0 + 1.0 = 2.0$
    - ✓ Maximum possible speedup
      - $T_\infty = 0.1 * 10.0 = 1.0$

Program Profiling

Major insights of Amdahl's law

- Substantial improvement to a major part of the system may not results in significant net speedup
- We must improve the speed of a very large fraction of the overall system, to significantly speed up the entire system

Summary

[Appendix] Program Profiling

■ Program profiling

- Profile report
  - Example) Flat profile

```
 %         cumulative   self              self      total
 time        seconds   seconds    calls   s/call    s/call  name

33.34          0.02      0.02      7208     0.00      0.00   open
16.67          0.03      0.01       244     0.04      0.12   offtime
16.67          0.04      0.01         8     1.25      1.25   memccpy
16.67          0.05      0.01         7     1.43      1.43   write
16.67          0.06      0.01                                mcount
 0.00          0.06      0.00       236     0.00      0.00   tzset
 0.00          0.06      0.00       192     0.00      0.00   tolower
 0.00          0.06      0.00        47     0.00      0.00   strlen
 0.00          0.06      0.00        45     0.00      0.00   strchr
 0.00          0.06      0.00         1     0.00     50.00   main
 0.00          0.06      0.00         1     0.00      0.00   memcpy
 0.00          0.06      0.00         1     0.00     10.11   print
 0.00          0.06      0.00         1     0.00      0.00   profil
 0.00          0.06      0.00         1     0.00     50.00   report

...
```

https://sourceware.org/binutils/docs/gprof/Flat-Profile.html#Flat-Profile

[Appendix] Program Profiling

Program profiling

- Profile report
  - Example) Call graph

```
index  %time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02     8+472       <cycle 2 as a whole> [4]
                0.01        0.02   244+260           offtime <cycle 2> [7]
                0.00        0.00
```

https://sourceware.org/binutils/docs/gprof/Call-Graph.html#Call-Graph