Computer Organization: A Programmer's Perspective

Profiling

Gal A. Kaminka
galk@cs.biu.ac.il

Profiling: Performance Analysis

Performance Analysis (“Profiling”)

- Understanding the run-time behavior of programs
- What parts are executed, when, for how long
- What parts require improvement, optimization

Profiling Programs

- Tools of the trade
- Granularity of profiling (modules, functions, instructions, ...)
- Performance measurement

Components of Performance

Run time

How long does it take to compute?

Memory

How much memory does it take?

<u>These are issues that affect the above components:</u>

Input/Output (I/O)

How much access to external devices, services?

System calls

Parallelization of tasks

Measurement Challenge

How Much Time Does Program X Require?

CPU time

- How many total seconds are used when executing X?
- Measure used for most applications
- Small dependence on other system activities

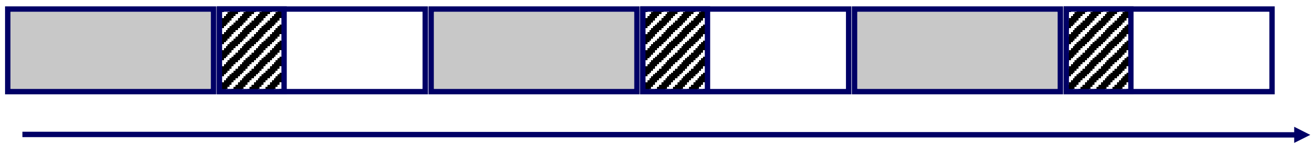
Actual ("Wall-Clock") Time

- How many seconds elapse between start and completion of X?
- Depends on system load, I/O times, etc.

Confounding Factors

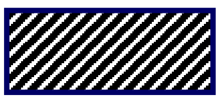
- How does time get measured?
- Many processes share computing resources
  - Transient effects when switching from one process to another
  - The effects of alternating among processes become noticeable

"Time" on a Computer System

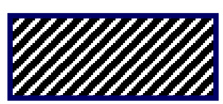


real (wall clock) time

 **= user time** *(time executing instructions in the user process)*

 **= system time** *(time executing instructions in kernel on behalf of user process)*

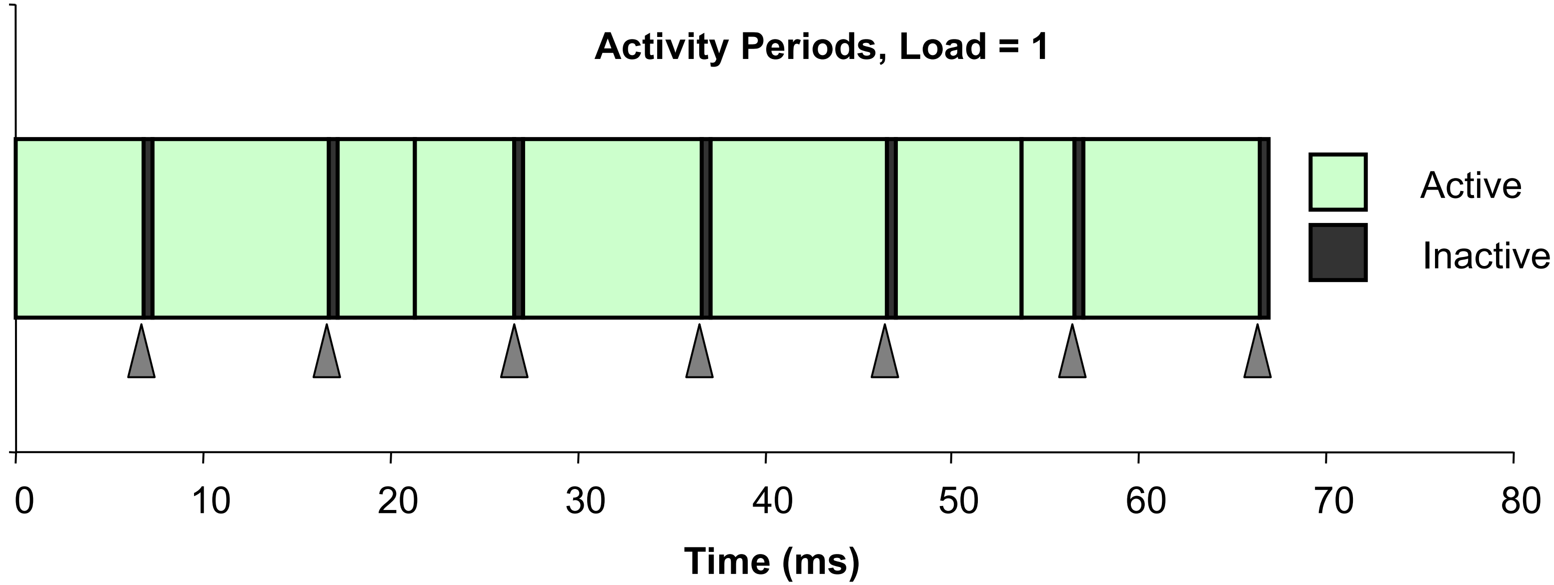
 **= some other user's time** *(time executing instructions in different user's process)*

 **+**  **+**  **= real (wall clock) time**

We will use the word "time" to refer to user time.

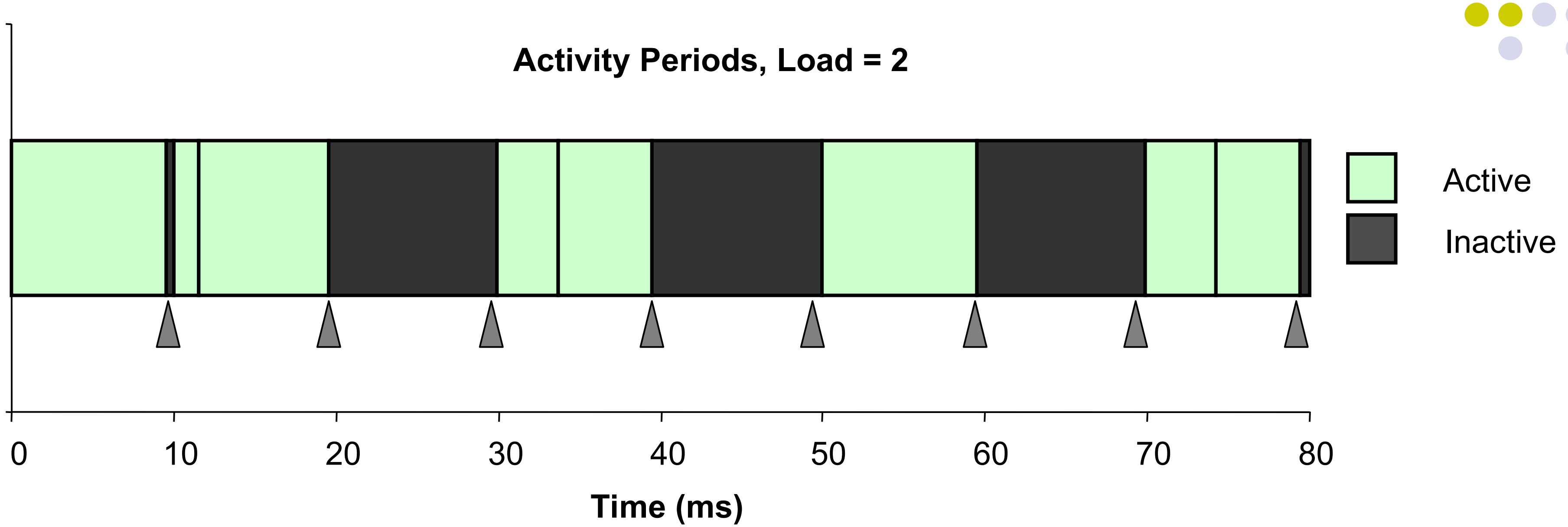
 **cumulative user time**

Activity Periods: Light Load



- Most of the time spent executing one process
- Periodic interrupts every 10ms
  - Interval timer
  - Keep system from executing one process to exclusion of others
- Other interrupts
  - Due to I/O activity
- Inactivity periods
  - System time spent processing interrupts
  - ~250,000 clock cycles

Activity Periods: Heavy Load



Sharing processor with one other active process

From perspective of this process, system appears to be "inactive" for ~50% of the time

- Other process is executing

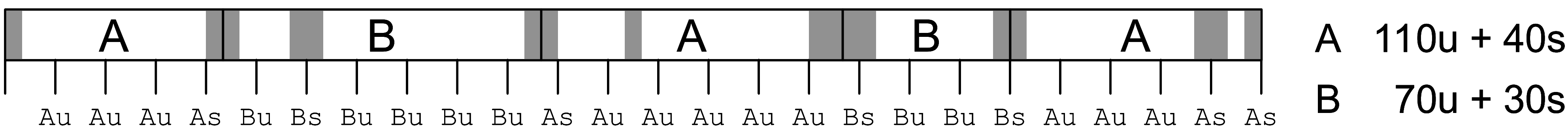
Interval Counting

OS Measures Run-times Using Interval Timer

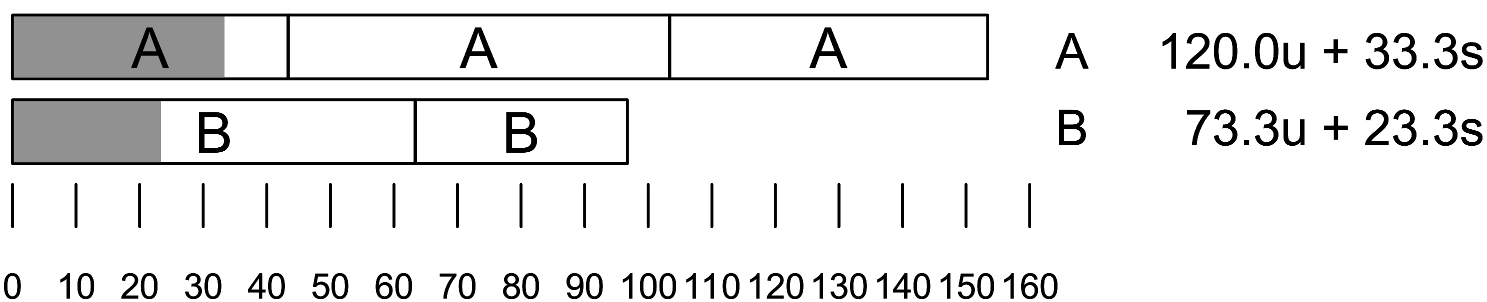
- Maintain 2 counts per process
  - User time
  - System time
- Each time get timer interrupt, increment counter for executing process
  - User time if running in user mode
  - System time if running in kernel mode

Interval Counting Example

(a) Interval Timings



(b) Actual Times



Unix `time` Command

(here, timing a “make osevent” command)

```
> time make osevent
gcc -O2 -Wall -g  -march=i486 -c clock.c
gcc -O2 -Wall -g  -march=i486 -c options.c
gcc -O2 -Wall -g  -march=i486 -c load.c
gcc -O2 -Wall -g  -march=i486 -o osevent osevent.c . . .
0.820u 0.300s 0:01.32 84.8%     0+0k 0+0io 4049pf+0w
>
```

0.82 seconds user time
- 82 timer intervals

0.30 seconds system time
- 30 timer intervals

1.32 seconds wall time

84.8% of total was used running these processes
- (0.82+0.3)/1.32 = .848

Unix `time` Command
(here, timing a "make osevent" command)

```
> time make osevent
gcc -O2 -Wall -g  -march=i486 -c clock.c
gcc -O2 -Wall -g  -march=i486 -c options.c
gcc -O2 -Wall -g  -march=i486 -c load.c
gcc -O2 -Wall -g  -march=i486 -o osevent osevent.c . . .
0.820u 0.300s 0:01.32 84.8%     0+0k 0+0io 4049pf+0w
>
```

0.82 seconds user time
- 82 timer intervals

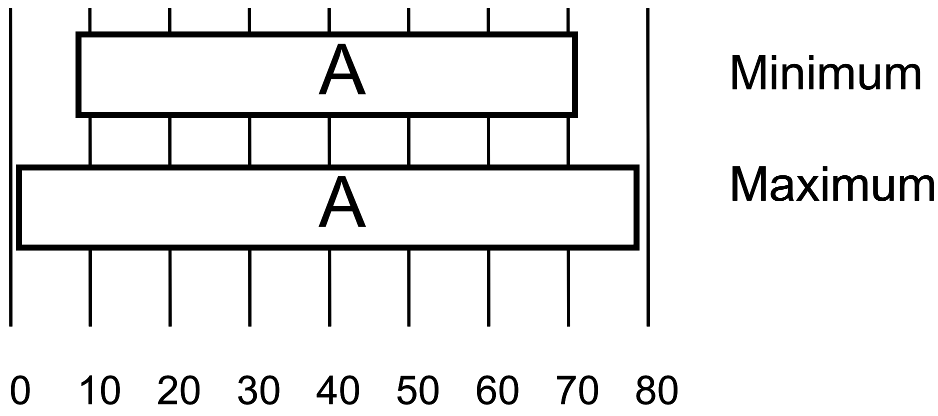
0.30 seconds system time
- 30 timer intervals

1.32 seconds wall time

84.8% of total was used running these processes
- (0.82+0.3)/1.32 = .848

`time` tells us where the CPU time is spent:
Our code, the system (I/O), or elsewhere

Accuracy of Interval Counting



Computed time = 70ms

Min Actual = 60 + ε

Max Actual = 80 – ε

- Average Case Analysis
  - Over/underestimates tend to balance out
  - As long as total run time is sufficiently large
    - Min run time ~1 second
    - 100 timer intervals
  - Consistently miss 4% overhead due to timer interrupts

The 90/10 Rule of Thumb

90% of execution time is in 10% of code

Lesson:

- Find the 10% that really count!
- Let the compiler worry about the rest

Important:

- First make program work correctly
- Make sure easy to maintain
- Then optimize

Priority depends on project size, scope, maturity

Profiling Within Our Code

Profiling at sub-program level

<u>We can measure execution time at:</u>

All functions of a program

- Flat statistics
- Call context statistics

Specific function

- Specific instructions, operations
- Memory use, system calls, etc.

Profiling modules (functions)

- Profilers: Tools used to measure run-time performance
  - Frequency and duration of function execution
  - Memory use
- Input:
  - Events (object creation/deletion, thread state, method calls)
  - Instruction counts (how many CPU instructions ran), clock cycles
    - Typically (sampled counts, not accurate)
  - Counters (frequency and duration)
    - Instrumentation
- Output:
  - Execution trace
  - Statistics

Profile types

- Flat: time spent in each function
  - % time (out of total running time)
  - # of calls made to this function
  - Average, maximum, minimum execution-time per call
    - - Self
    - - Including descendants in call graph

- Call-graph: performance depending on call stack
  - e.g., duration depending on whom was caller, what was passed

Examples of profilers

- `gprof` (compiler-assisted instrumentation)
  - Compile (and link) with "-pg" flag
  - During run-time, program will create file "gmon.out"
  - "`gprof <exec> > report.txt`" will generate report
  - Flat and call-graph run-time profiling
- `valgrind` (run-time instrumentation)
  - run "valgrind <exec>", get a report
  - Several tools:
    - Memory leak checker, other memory bugs
    - Cache use profiling
    - Heap memory profiling (who is allocating memory)

Examples of profilers

- `cProfile` (python)
  - e.g., "`python -m cProfile` -o *`prg.prof`* *`prg.py`*"
  - Flat and call-graph run-time profiling
- `Py-spy` (python)
  - e.g., "`py-spy record -o` *`output.svg`* `-pid` *`pid`*"
- `Visualvm, JDK Mission Control, glowroot` (Java)
  - Also many built into different IDEs
- Every professional programmer needs to know profilers!

Example *flat* gprof output

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.50	0.23	0.23				main
17.50	0.30	0.07	3	23.33	23.33	count4()
7.50	0.33	0.03	2	15.00	38.33	count3()
7.50	0.36	0.03	1	30.00	30.00	count1()
5.00	0.38	0.02	1	20.00	20.00	count()
5.00	0.40	0.02	1	20.00	58.33	count2()

% time spent out of total

Cumulative and self seconds spent

# of calls, msec per call (self and total)

Example *flat* gprof output

```
 %     cumulative   self              self     total
time     seconds   seconds  calls  ms/call  ms/call    name
57.50      0.23      0.23                               main
17.50      0.30      0.07       3    23.33    23.33  count4()
 7.50      0.33      0.03       2    15.00    38.33   count3()
 7.50      0.36      0.03       1    30.00    30.00   count1()
 5.00      0.38      0.02       1    20.00    20.00   count()
 5.00      0.40      0.02       1    20.00    58.33   count2()
```

% time spent out of total

Cumulative and self seconds spent

# of calls, msec per call (self and total)

Example *flat* gprof output

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.50	0.23	0.23				main
17.50	0.30	0.07	3	23.33	23.33	count4()
7.50	0.33	0.03	2	15.00	38.33	count3()
7.50	0.36	0.03	1	30.00	30.00	count1()
5.00	0.38	0.02	1	20.00	20.00	count()
5.00	0.40	0.02	1	20.00	58.33	count2()

% time spent out of total

Cumulative and self seconds spent

# of calls, msec per call (self and total)

Example *flat* gprof output

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.50	0.23	0.23				main
17.50	0.30	0.07	3	23.33	23.33	count4()
7.50	0.33	0.03	2	15.00	38.33	count3()
7.50	0.36	0.03	1	30.00	30.00	count1()
5.00	0.38	0.02	1	20.00	20.00	count()
5.00	0.40	0.02	1	20.00	58.33	count2()

% time spent out of total

Cumulative and self seconds spent

# of calls, msec per call (self and total)

Example *flat* gprof output

```
 %     cumulative   self              self     total
time     seconds   seconds  calls  ms/call  ms/call    name
57.50       0.23     0.23                                main
17.50       0.30     0.07       3    23.33    23.33   count4()
 7.50       0.33     0.03       2    15.00    38.33    count3()
 7.50       0.36     0.03       1    30.00    30.00    count1()
 5.00       0.38     0.02       1    20.00    20.00    count()
 5.00       0.40     0.02       1    20.00    58.33    count2()
```

% time spent out of total

Cumulative and self seconds spent

# of calls, msec per call (self and total)

Example *flat* cProfile output

[stackoverflow.com/questions/582336/how-can-you-profile-a-python-script]

```
 1007 function calls in 0.061 CPU seconds

ncalls  tottime  percall  cumtime  percall file:line#(function)

    1    0.000    0.000    0.061    0.061 <string>:1(<module>)
 1000    0.051    0.000    0.051    0.000 euler048.py:2(<lambda>)
    1    0.005    0.005    0.061    0.061 euler048.py:2(<module>)
    1    0.000    0.000    0.061    0.061 {execfile}
    1    0.002    0.002    0.053    0.053 {map}
    1    0.000    0.000    0.000    0.000 {method 'disable' ...}
objects}
    1    0.000    0.000    0.000    0.000 {range}
    1    0.003    0.003    0.003    0.003 {sum}
```

See: Python Profiling (Amjith Ramanujam on youtube)

- Shows also GUI tools and more tools, how to use, etc.
- Remember others exist

Code Profiling Example

- Task: Count *n*-gram frequencies in text document
  - Sorted list of words (1-gram) from most frequent to least
  - Also pairs (2-gram)
- Information retrieval, natural language processing

Shakespeare's most frequent words

29,801	the
27,529	and
21,029	I
20,957	to
18,514	of
15,370	a
14010	you
12,936	my
11,722	in
11,519	that

- Data Set
  - Collected works of Shakespeare
  - 946,596 total words, 26,596 unique

Code Profiling

Augment Executable Program with Timing Functions

- Computes (approximate) amount of time spent in each function
- Also maintains counter for each function indicating number of times called

Using

```
gcc -O2 -pg prog. -o prog
```

```
./prog
```

- Executes in normal fashion, but also generates file `gmon.out`

```
gprof prog
```

- Generates profile information based on `gmon.out`

Implementation

- Steps
  - Convert strings to lowercase
  - Apply hash function
  - Read words and insert into hash table
    - Mostly list operations
    - Maintain counter for each unique word
  - Sort results
- Initial implementation
  - Sort: insertion sort(?)
  - List insertion: at end of list (via recursive call)
  - Hash: sum of characters in word, modulo (%) table size
  - Convert to lower:

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

Profiling Results

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
86.60	8.21	8.21	1	8210.00	8210.00	sort_words
5.80	8.76	0.55	946596	0.00	0.00	lower1
4.75	9.21	0.45	946596	0.00	0.00	find_ele_rec
1.27	9.33	0.12	946596	0.00	0.00	h_add

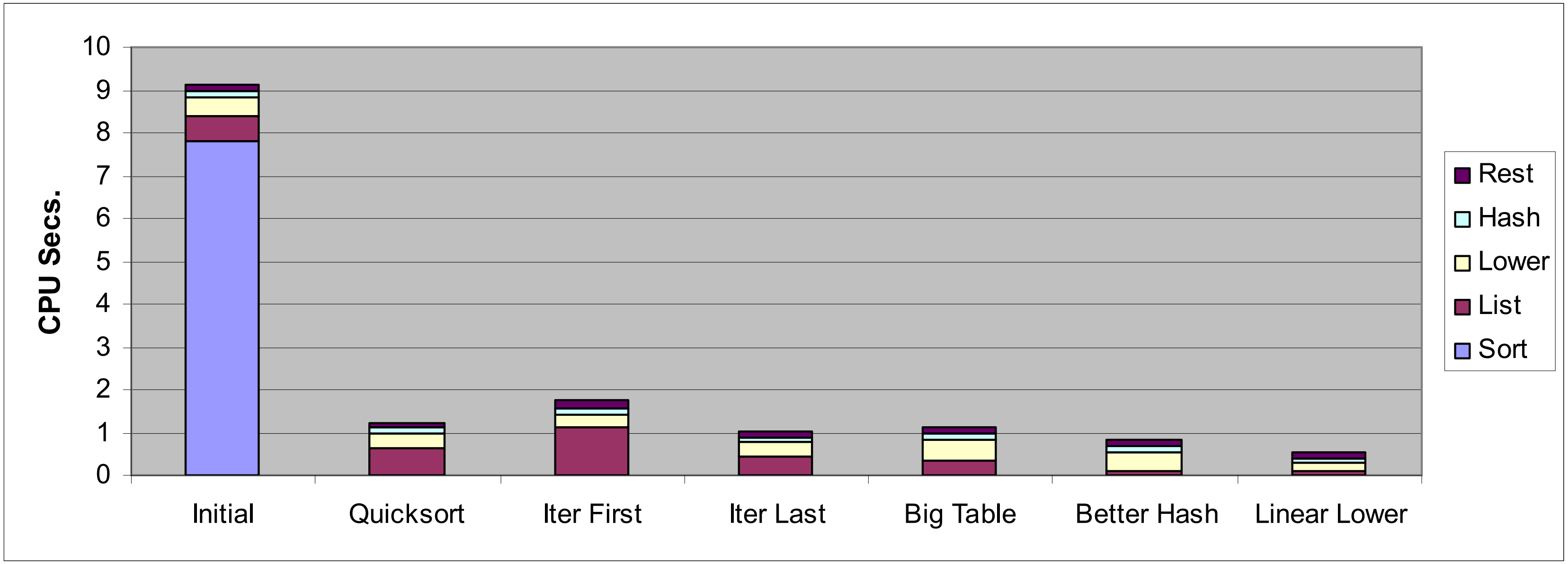
Call Statistics

- Number of calls and cumulative time for each function

Performance Limiter

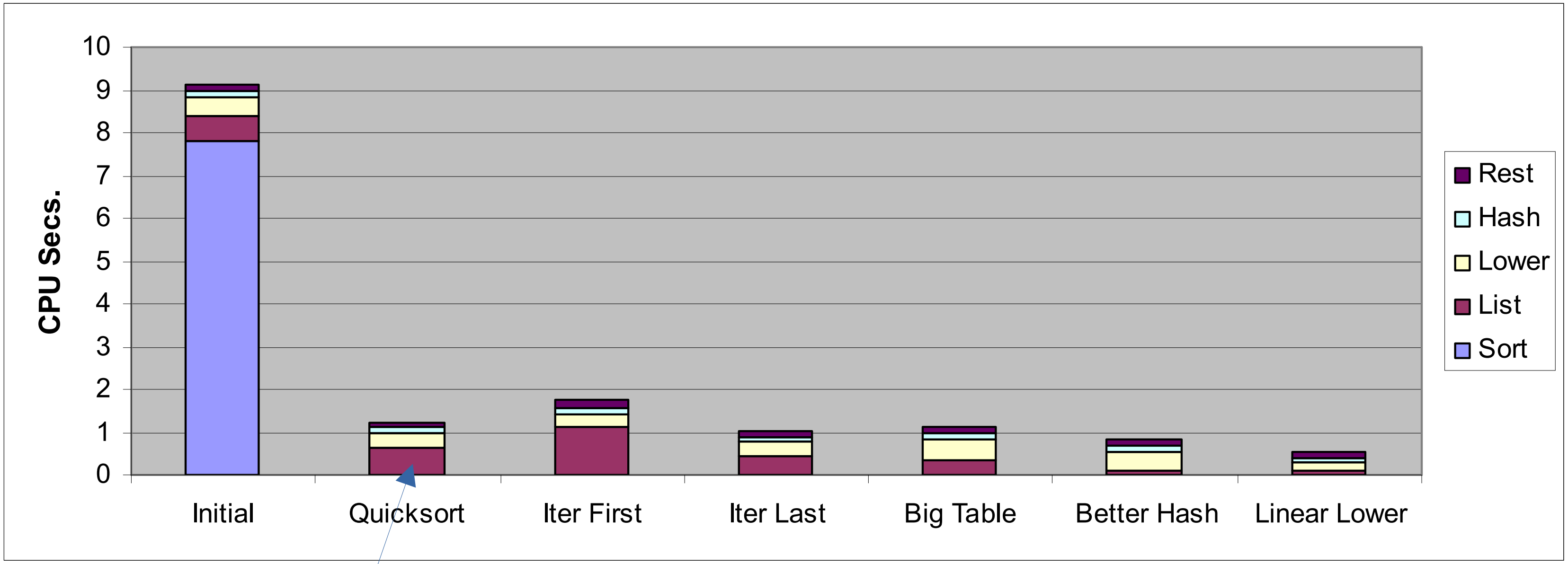
- Using inefficient sorting algorithm
- Single call uses 87% of CPU time

Code Optimizations



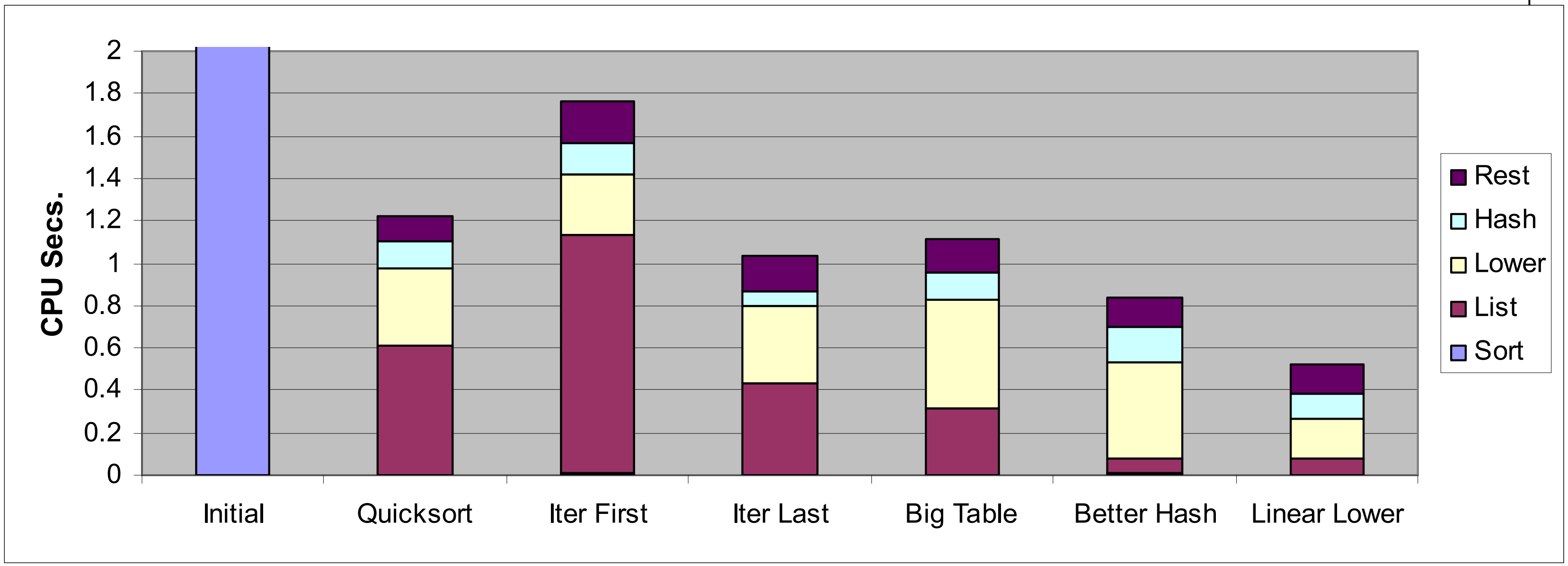
- First step:
  - Use more efficient sorting function
  - Library function `qsort`

Code Optimizations



- First step:
  - Use more efficient sorting function
  - Library function `qsort`
- Now list operations main issue

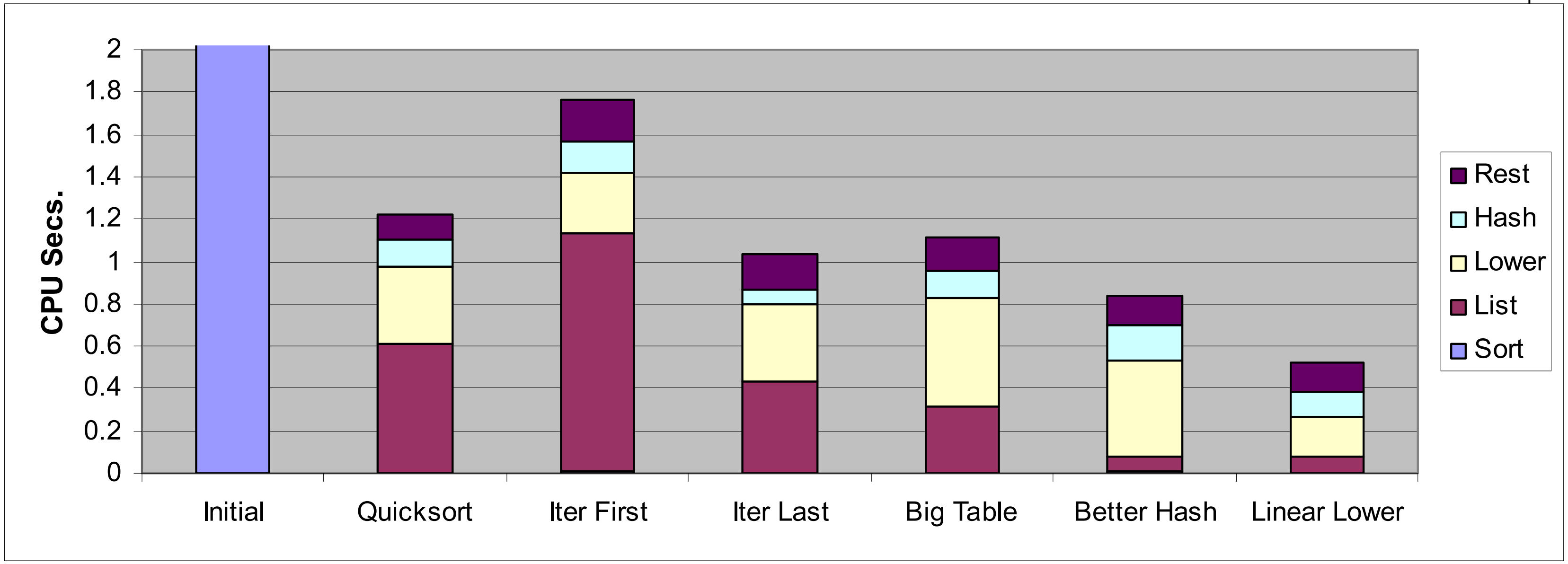
Further Optimizations



<u>Improve list operations:</u>

- Iteration (loop) instead of recursion
- <u>Iter First</u>: insert elements into first place of linked list
  - Causes code to slow down
- <u>Iter Last</u>: insert elements at end of list
  - Much better. Why?

Further Optimizations

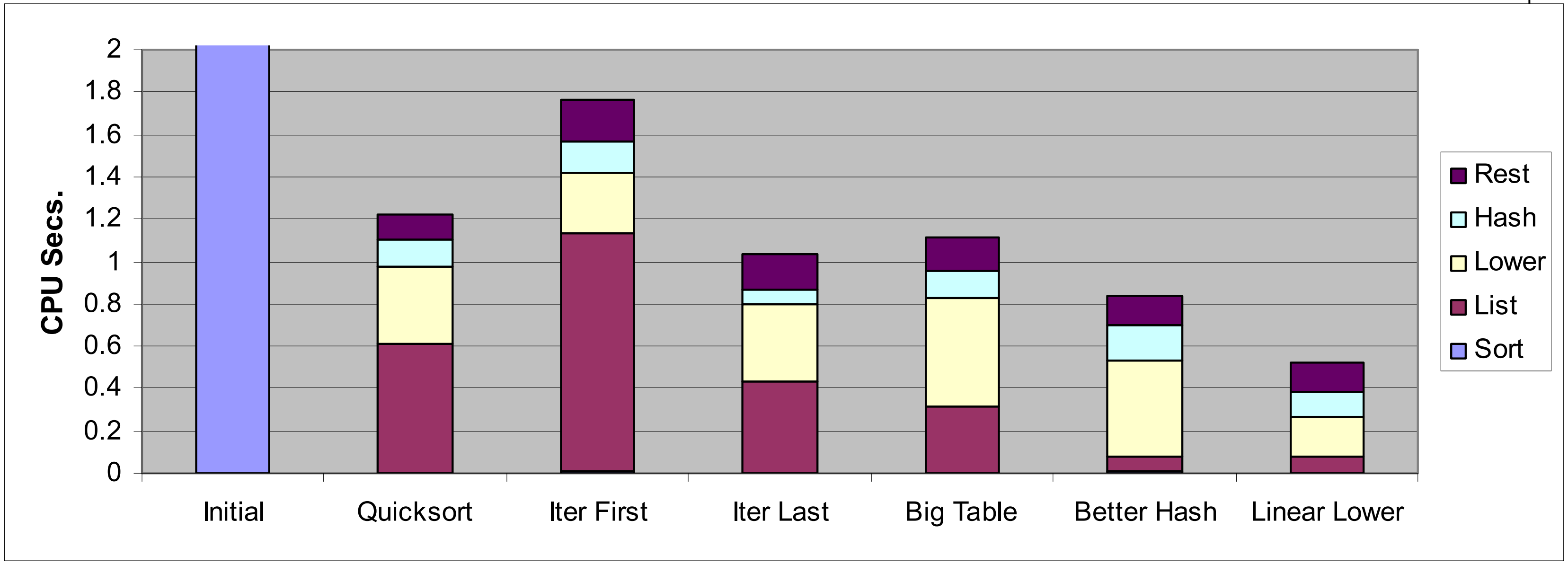


<u>Improve list operations:</u>

- Iteration (loop) instead of recursion
- <u>Iter First</u>: insert elements into first place of linked list
  - Causes code to slow down
- <u>Iter Last</u>: insert elements at end of list
  - Much better. Why?

Tend to place most common words at front of list

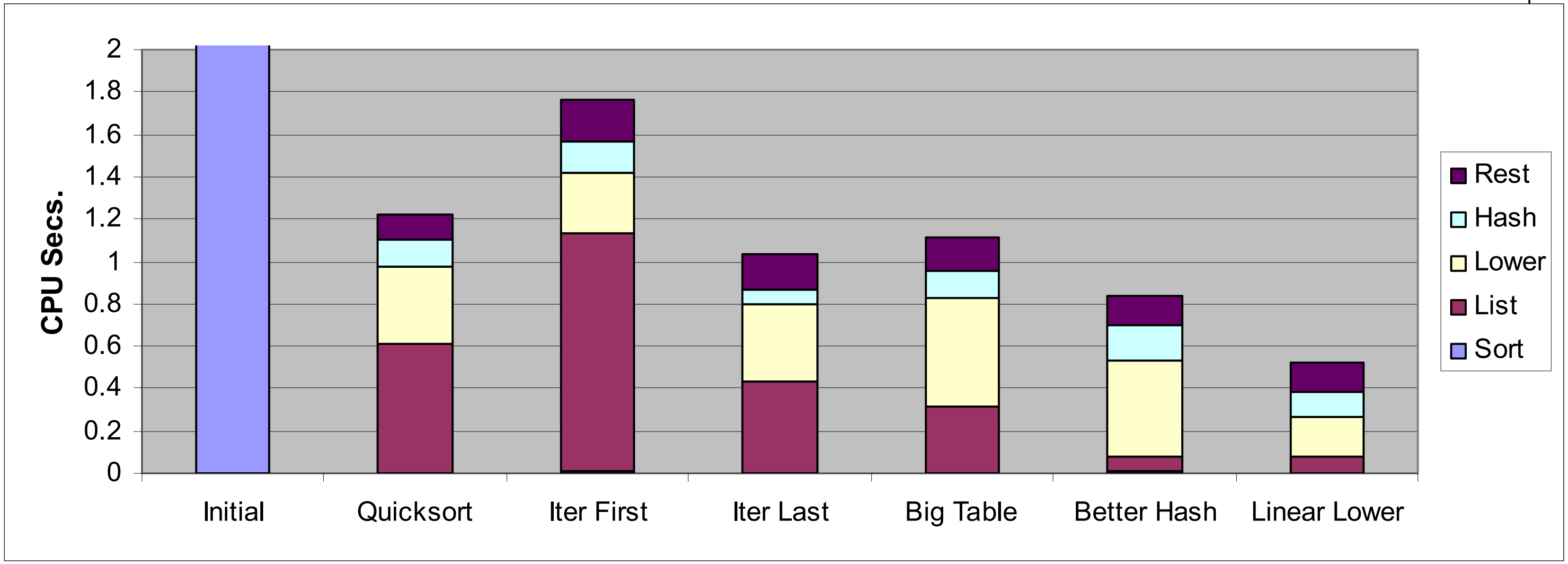
Further Optimizations



Hashing

- Big table: Increase number of hash buckets
- Better hash: Use more sophisticated hash function

Further Optimizations



<u>Lower</u>

- Move strlen out of loop:

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

Implementation matters (even on fast machines)

- For 1-gram (single words), from 9.3 to 0.5 (X ~20 speedup)
  - This was on an old 32 bit machine
  - Does it really matter?
- On i7, 16GB, bought early in 2020:
  - 1-gram speedup: from 0.26 to 0.02 (~13)
  - 2-gram speedup: from 238.14 to 0.15 (~1587.6)

- Profiling is standard, common practice
  - Among professionals
  - Very powerful tools

Observations

Benefits

- Helps identify performance bottlenecks
- Especially within complex system with many components

Limitations

- Only shows performance for data tested
  - e.g., linear lower did not show big gain, since words are short
  - Quadratic inefficiency could remain lurking in code
- Timing mechanism fairly crude
  - Only works for programs that run for > 3 seconds

Do some self-studying!

Both gprof and valgrind are extremely powerful tools

Many options, many features

Take the time to study them

They can save you many many hours of work

- www.valgrind.org
- “man gprof”, “man valgrind”
- Google for tutorials
- ....

Things NOT to do

Reduce number of lines in code, write unreadable code

- There is no direct relation between # lines and execution time
- Compare:

```
for (int i=0; i<5; i++) a[i] = i;
```

to:

```
a[0]=0; a[1]=1; a[2]=2; a[3]=3; a[4]=4
```

Optimize as you go

- Invest effort without thought
- Often makes code difficult to maintain, re-use

Guess at where to spend effort

- Instead: Measure! **Profile!** Don't guess.