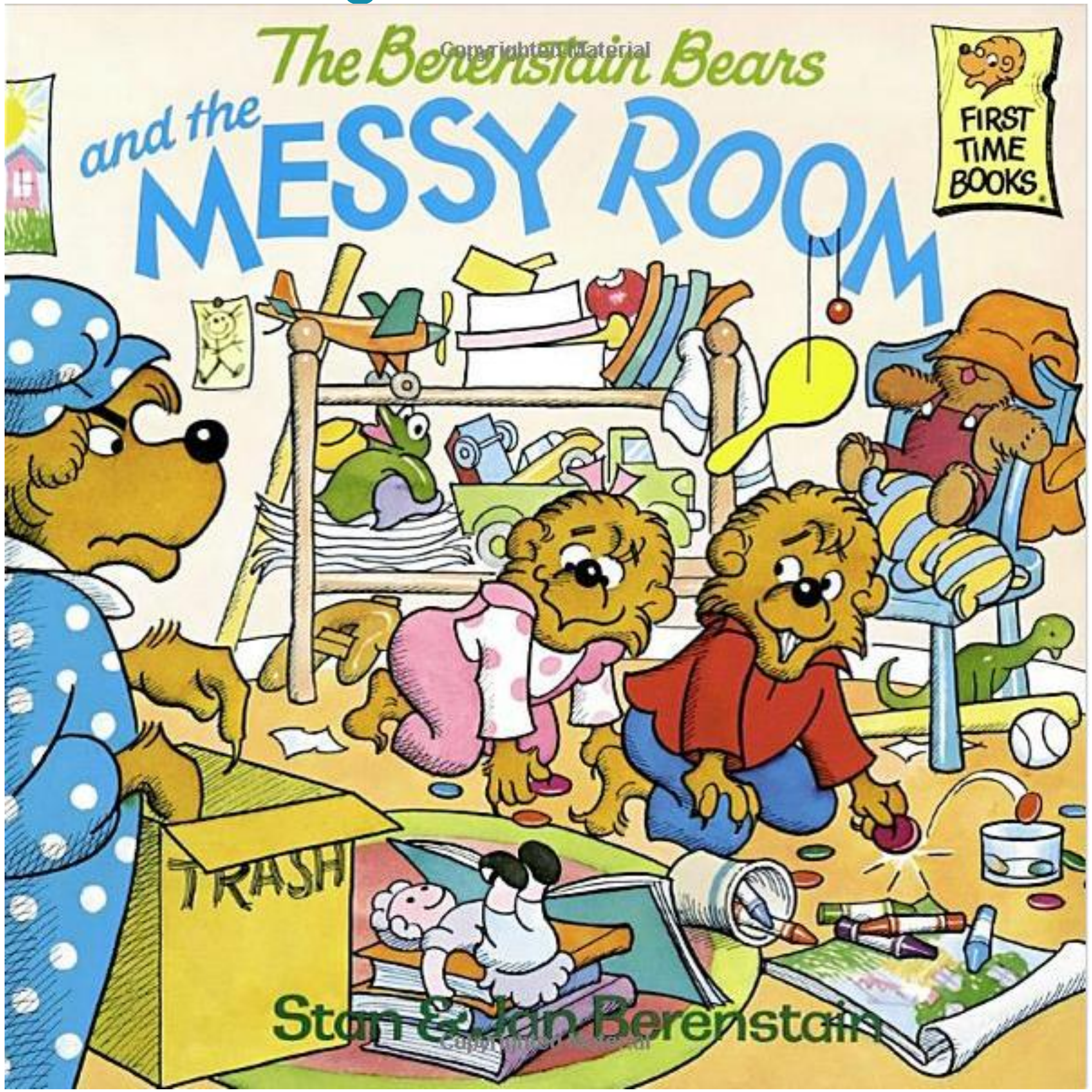
BETTER DATA ORGANIZATION FOR FASTER SEARCHES

Reduce Search Complexity with Better Data Organization

- Reduce the number of items to examine, by …
- Organizing the data better to find the right items faster, maybe by…
- Using a better data structure which supports better search algorithms



S. Berenstain & J. Berenstain, Random House,1983

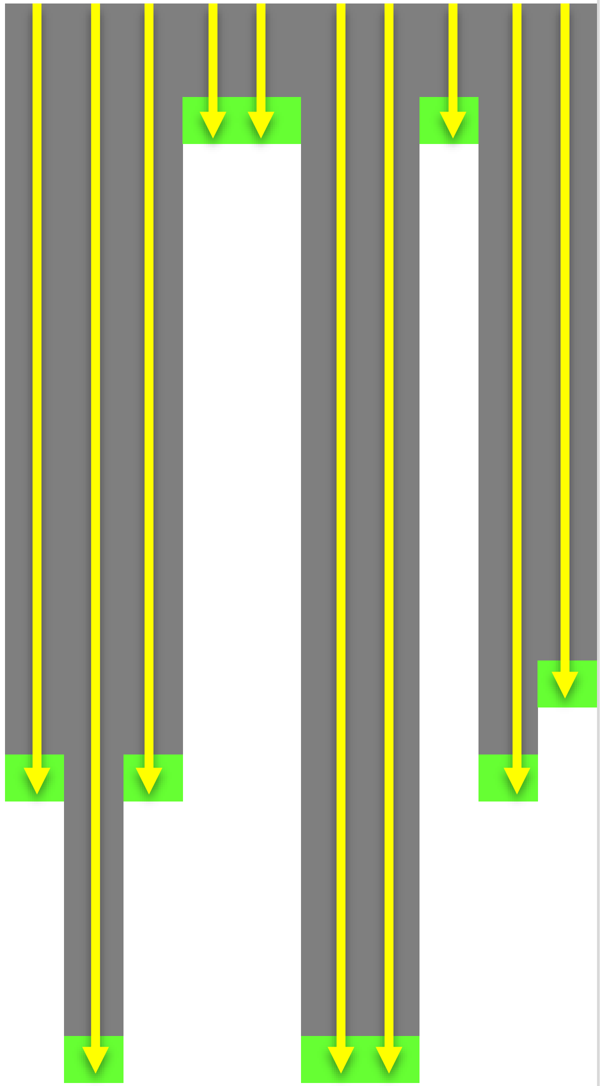
Optimizing the Execution Time Profiler

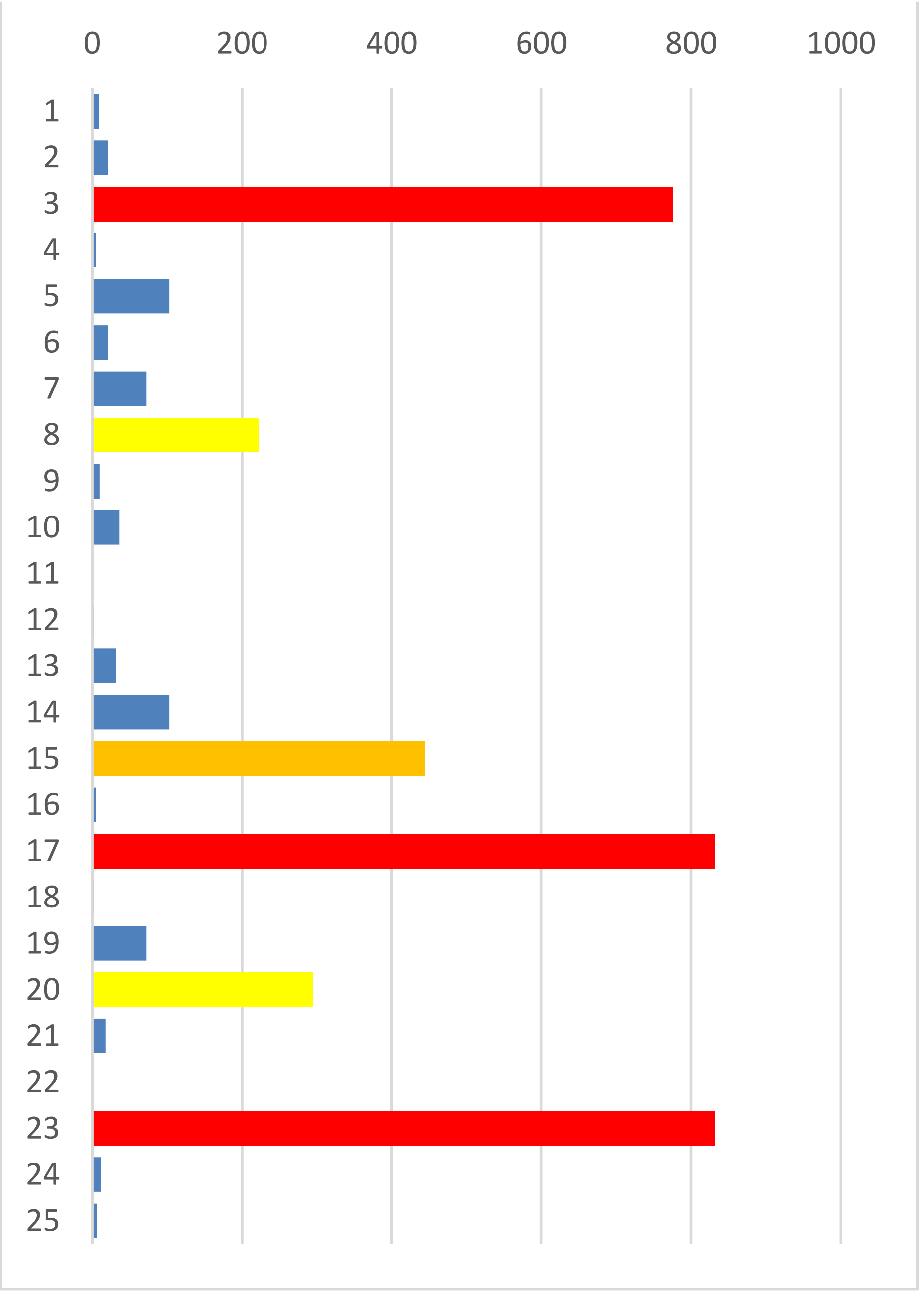
- Periodic PC-sampling ISR
  - Determines return address
  - Searches table with return address for region number
  - Increments execution count for that region
- RegionTable array
  - Holds start, end addresses of each region to monitor
- Search function
  - Searches for region with addresses bounding the search address (start address ≤ search value ≤ end address)
  - RegionTable is an ***array*** used as a ***list*** (sequential access starting at element 0)
- Execution time performance
  - Table has n elements
  - On average, search half of the elements in table (n/2)
  - => complexity is linear (O(n))
    - 2x elements => 2x average execution time
- Slow execution slows down the program, limits practical sampling frequency

```
const REGION_T RegionTable[] = {
  {0x000000d5, 0x0000014c, "SystemInit"}, // 0
  {0x0000014d, 0x0000022c, "SystemCoreClockUpdate"}, // 1
  {0x00000261, 0x00000268, "Reset_Handler"}, // 2
  {0x00000269, 0x0000026a, "NMI_Handler"}, // 3
  {0x0000026b, 0x0000026c, "HardFault_Handler"}, // 4
  {0x0000026d, 0x0000026e, "SVC_Handler"}, // 5
  {0x0000026f, 0x00000270, "PendSV_Handler"}, // 6
  {0x00000271, 0x00000272, "SysTick_Handler"}, // 7
  {0x0000027d, 0x000002ee, "main"}, // 8
  {0x00000305, 0x00000356, "Init_RGB_LEDs"}, // 9
  {0x00000357, 0x0000038a, "Control_RGB_LEDs"}, // 10
  {0x000003a1, 0x000003dc, "Init_Profiling"}, // 11
  {0x000003dd, 0x000003e4, "Disable_Profiling"}, // 12
  {0x000003e5, 0x000003ec, "Enable_Profiling"}, // 13
  {0x000003ed, 0x000003ee, "Clear_Lower_Screen"}, // 14
  {0x000003ef, 0x000003f0, "Print_Results"}, // 15
  {0x00000411, 0x0000049a, "PIT_IRQHandler"}, // 16
  {0x0000049b, 0x000004e6, "Init_PIT"}, // 17
  {0x000004e7, 0x000004f2, "Start_PIT"}, // 18
  {0x000004f3, 0x000004fe, "Stop_PIT"}, // 19
  {0x000004ff, 0x00000550, "Init_PWM"}, // 20
  {0x00000551, 0x0000056a, "Set_PWM_Values"}, // 21
  {0x000005c1, 0x000005ec, "__aeabi_uidivmod"}, // 22
  {0x000005ed, 0x0000068e, "__aeabi_fadd"}, // 23
  {0x0000068f, 0x00000696, "__aeabi_fsub"}, // 24
  {0x00000697, 0x0000069e, "__aeabi_frsub"}, // 25
  {0x0000069f, 0x00000718, "__aeabi_fmul"}, // 26
  {0x00000719, 0x00000730, "__ARM_scalbnf"}, // 27
  {0x00000731, 0x00000746, "__aeabi_i2f"}, // 28
  {0x00000747, 0x00000756, "_float_round"}, // 29
  {0x00000757, 0x000007ca, "_float_epilogue"}, // 30
  {0x000007cb, 0x00000846, "__aeabi_fdiv"}, // 31
  {0x00000847, 0x00000884, "_frnd"}, // 32
```

Performance Analysis

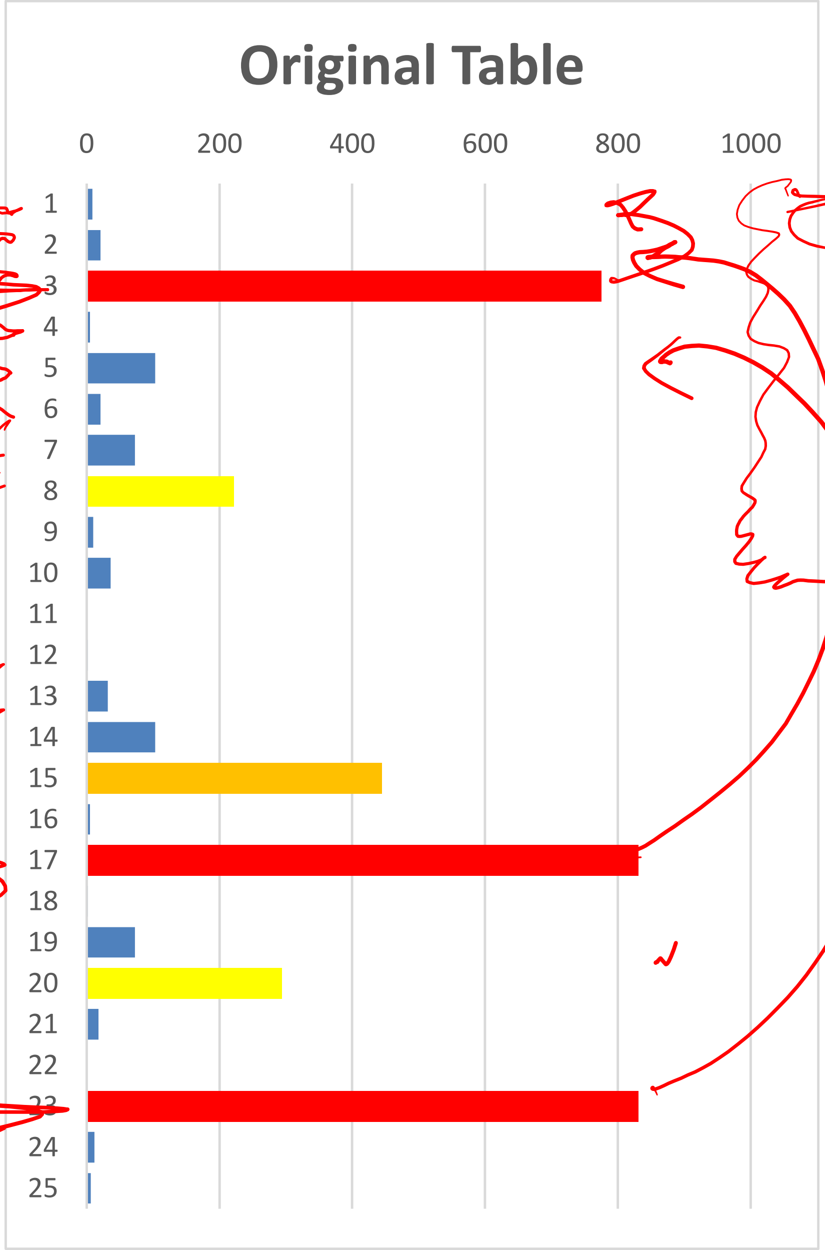
- Performance estimate
  - Table has n elements
  - Start at first element and go down until match found
  - On average, search half of the elements in table (n/2)
  - => complexity is linear (O(n))
    - 2x elements => 2x average execution time
- Slows down system, limits practical sampling frequency

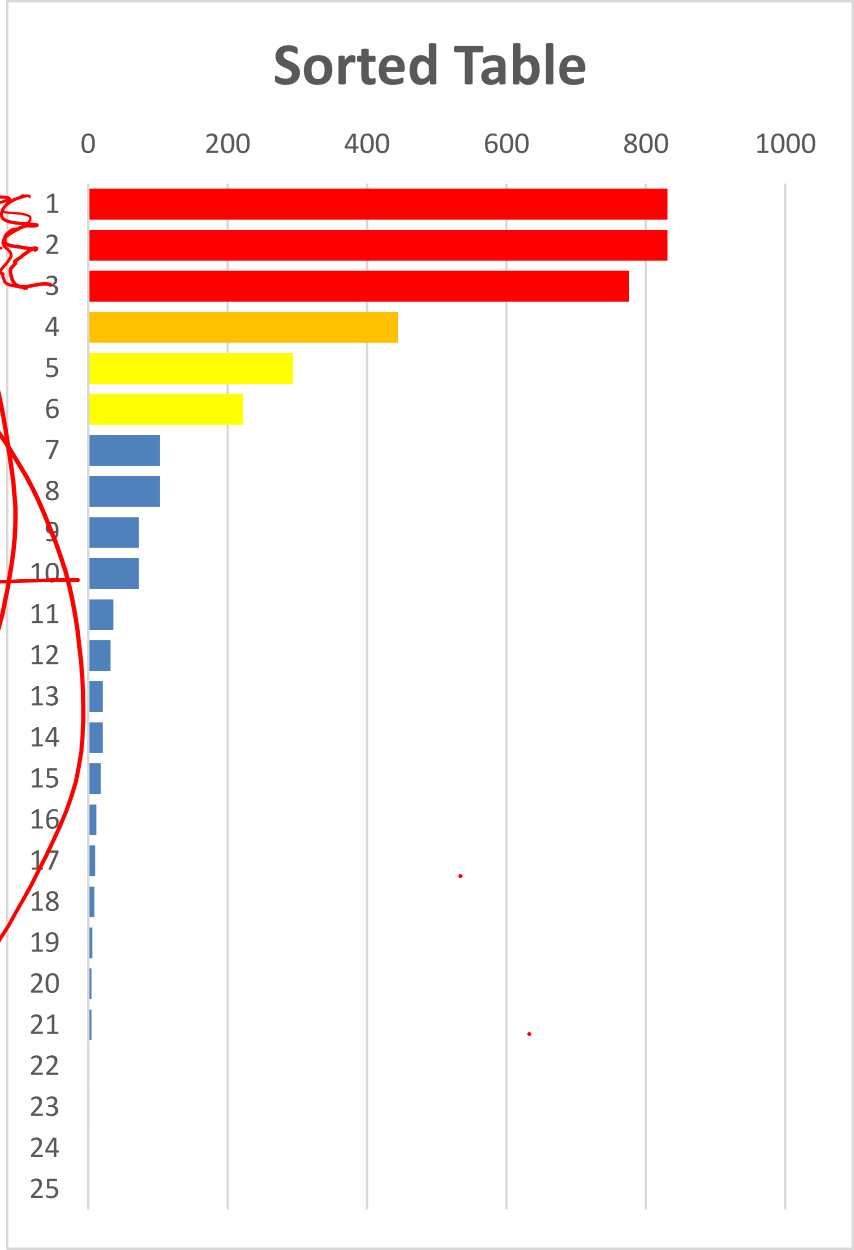




Optimization 1: Sort Table by Frequency

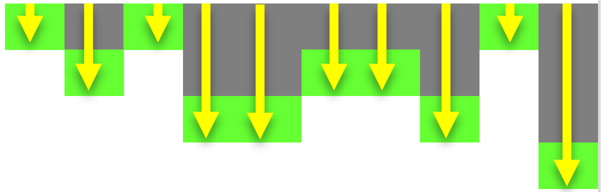
- Use estimated or initial profile information to generate a new region table sorted with *most frequent ("hot") regions at start*
- Reduces number of comparisons needed
- Must be done at compile time, after getting initial profile data

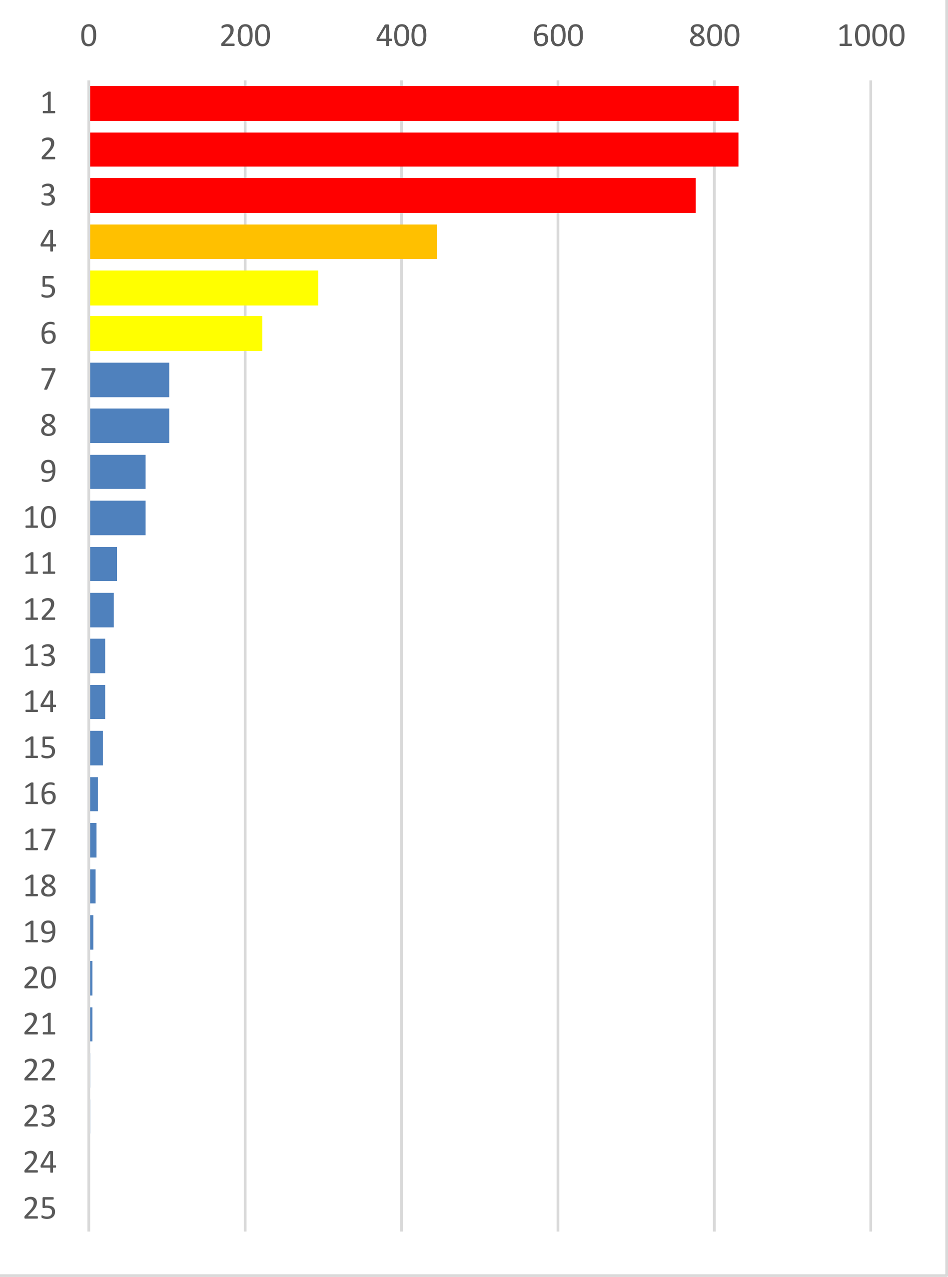




Optimization 1: Sort Table by Frequency

- Use estimated or initial profile information to generate a new region table sorted with *most frequent (“hot”) regions at start*
- Reduces number of comparisons needed
- Could be done…
  - … at compile time, after getting initial profile data
  - … adaptively, as program runs





Example: Changing the Region Table Layout

	RegionCount	RegionTable
[23]	1345	"__aeabi_fadd"}, // 23
[24]	20	"__aeabi_fsub"}, // 24
[25]	43	"__aeabi_frsub"}, // 25
[26]	1413	"__aeabi_fmul"}, // 26
[27]	46	"__ARM_scalbnf"}, // 27
[28]	65	"__aeabi_i2f"}, // 28
[29]	205	"_float_round"}, // 29
[30]	446	"_float_epilogue"}, // 30
[31]	0	"__aeabi_fdiv"}, // 31
[32]	16	"_frnd"}, // 32
[33]	7	"__aeabi_f2iz"}, // 33
[34]	0	"__scatterload"}, // 34
[35]	53	"__aeabi_ui2f"}, // 35
[36]	111	"__ARM_common_ll_muluu"}, //
[37]	121	"__ARM_fpclassifyf"}, // 37
[38]	12	"__mathlib_flt_underflow"},
[39]	494	"__mathlib_rredf2"}, // 39
[40]	0	

- New table layout
  - 0: __aeabi_fmul
  - 1: __aeabi_fadd
  - 2: __mathlib_rredf2
  - 3: _float_epilogue
  - 4: _float_round
  - *et cetera*

GetRegions

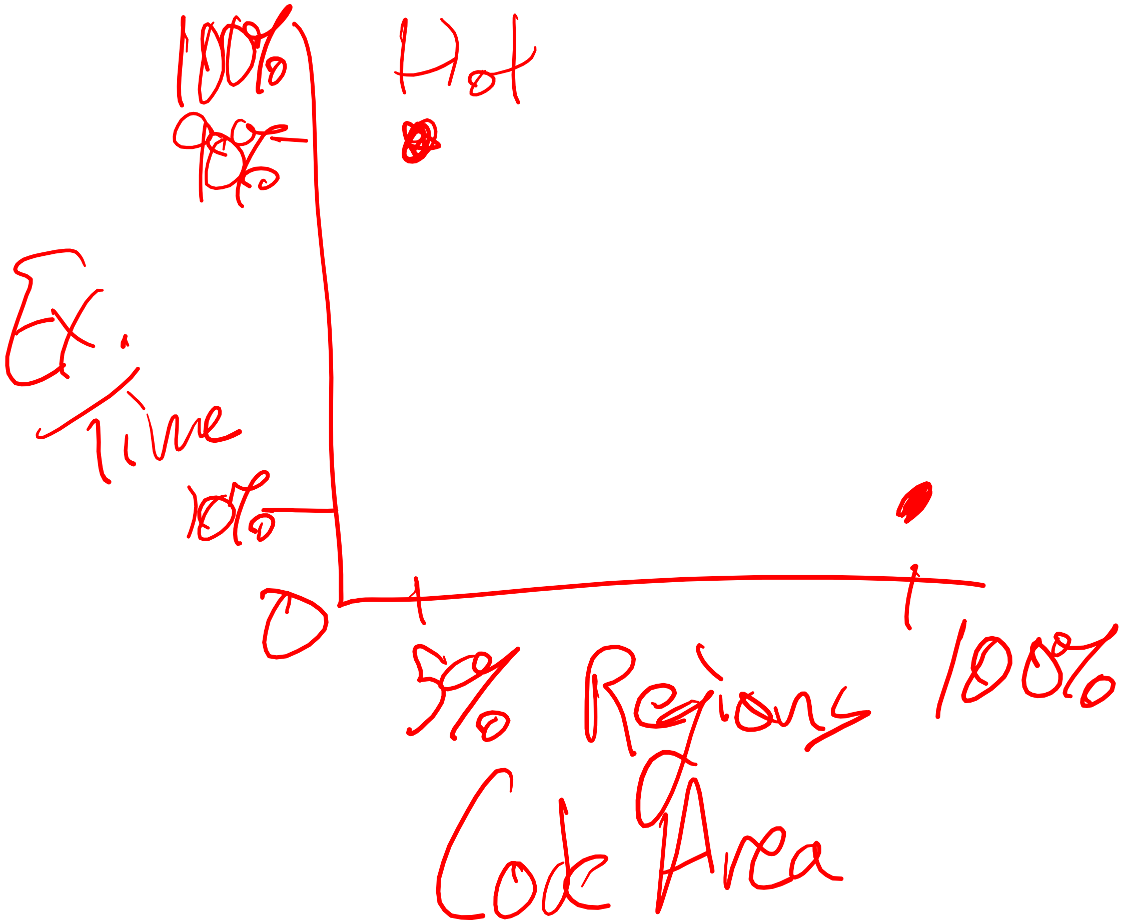
profile info.

- Static approach
  - Use tool in build process
- Dynamic approach
  - At run-time create and update a translation table

Estimating Performance Impact Without Profile Data

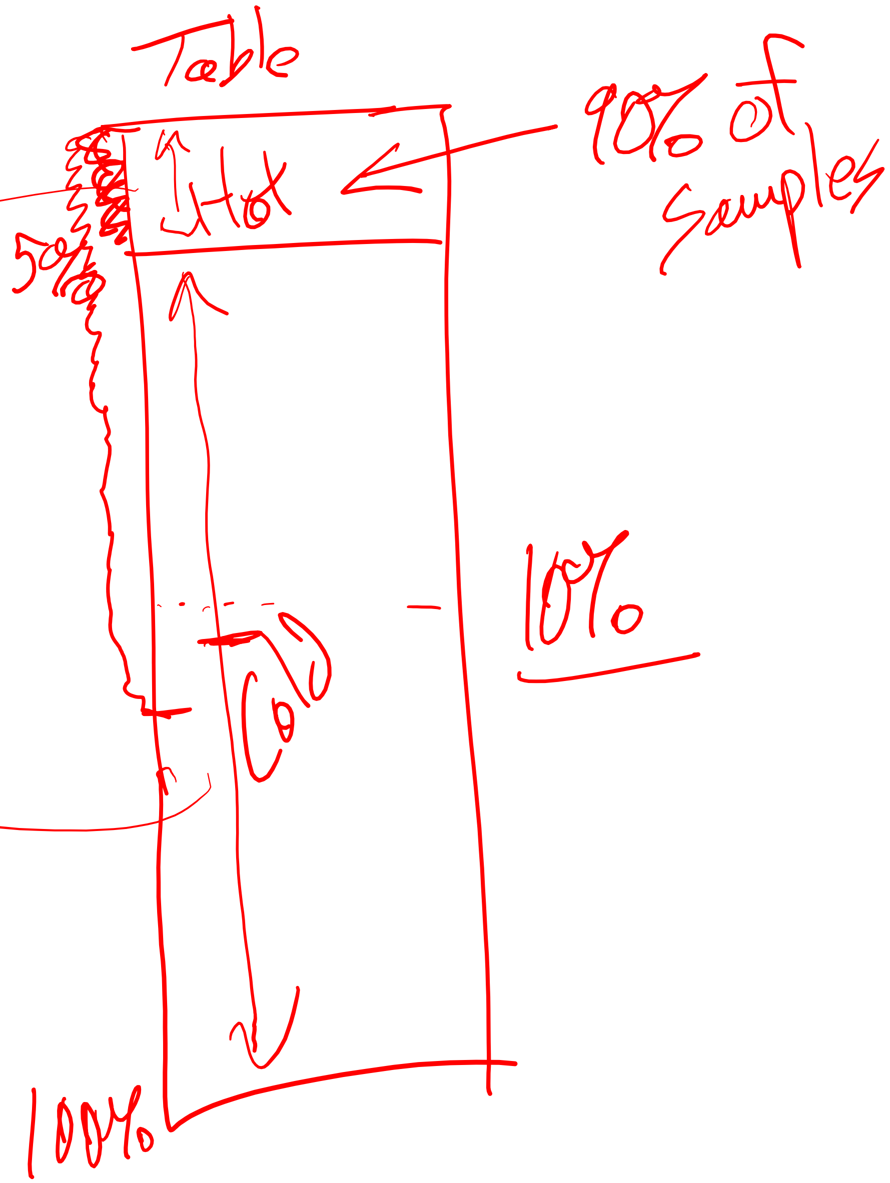
- Estimate hot region characteristics
  - Assume 5% of regions (hot regions) account for 90% of execution time
  - Uniform distribution within these 5% of regions

- Estimate cold region characteristics
  - Remaining 95% of regions account for remaining 10% of execution time
  - Uniform distribution within the remaining 95% of regions



Estimating Performance Impact Without Profile Data

- Estimate performance impact based on hot and cold regions
  - Hot regions
    - Most samples (90%) are in the first 5% of the table
    - $N_{iterations}(hot) = (S*90\%)*(R*5\%/2) = S*R*0.0225$
  - Cold regions
    - Remaining samples (10%) are in last 95% of the table
    - $N_{iterations}(cold) = (S*10\%)*(R*(0.95/2 + 0.05)) = S*R*0.0525$
- Overall performance
  - Improved layout: S*R*0.075
  - Original layout: S*R*0.5
  - Improvement in $N_{iterations}$ = 0.5/0.075 = 6.89x
- *Performance improves as more samples are concentrated in fewer regions*



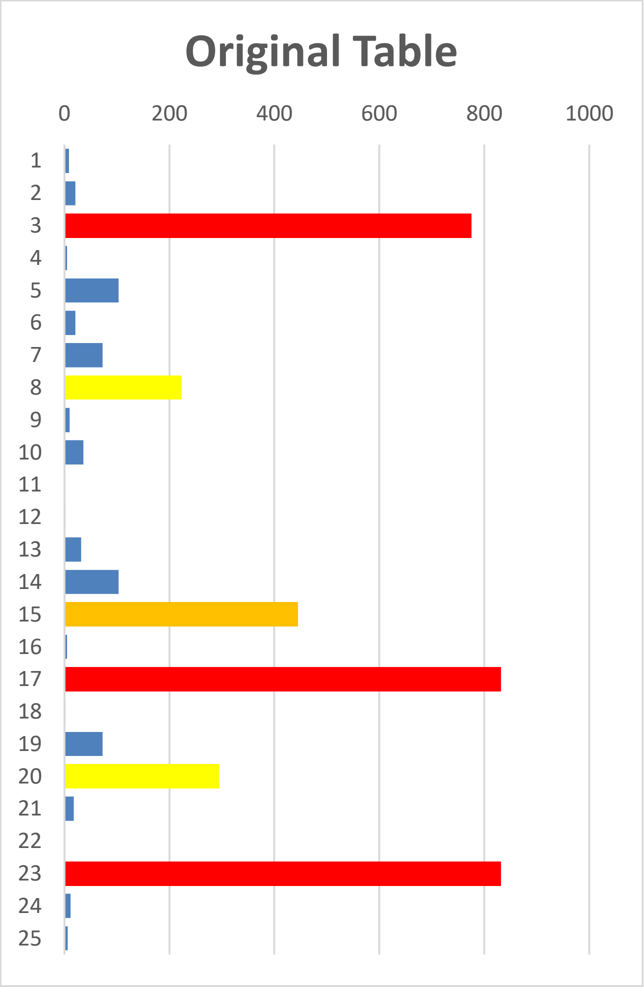
Estimating Performance Impact with Real Profile Data

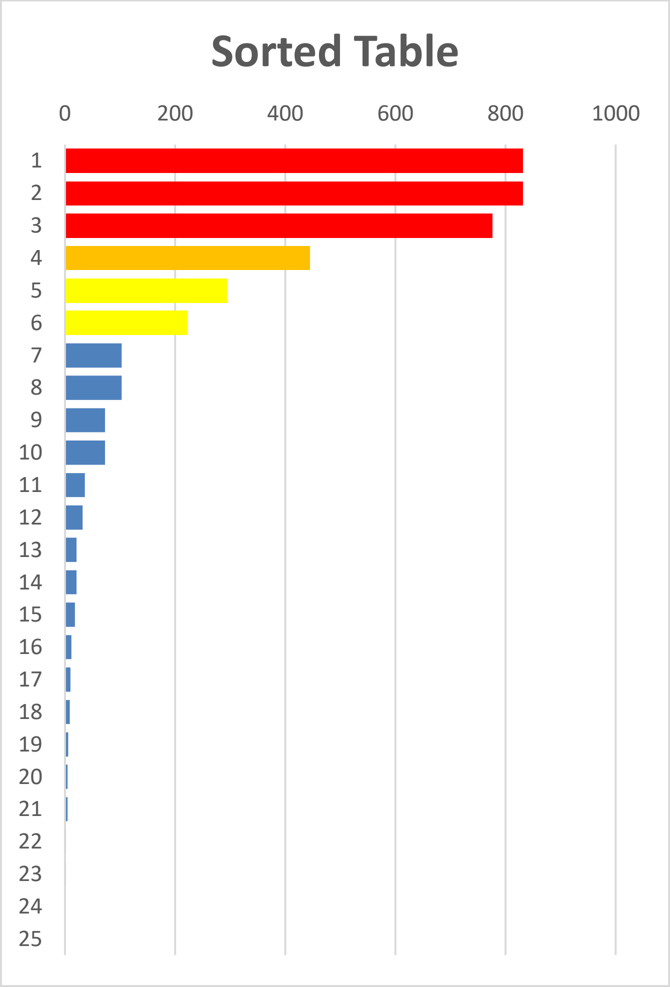
- More accurate calculation of profiling overhead
- Number of loop iterations required for region r
  - $N_{iterations}(r) = (r+1)*RegionCount[r]$
- Example: __aeabi_fmul is in position r=26
  - __aeabi_fmul lookups account for $N_{iterations}(26) = (26+1)*1413 = 38151$ iterations
- Swap __aeabi_fmul with region in position r=0
  - __aeabi_fmul lookups now account for $N_{iterations}(0) = (0+1)*1413 = 1413$ iterations
  - Old region in position 0 had RegionCount[0] = 0 samples, so it has no impact on performance
  - Result: we saved (38151 + 0) – (1413 + 0) = 36738 loop iterations
- Repeat this for all regions in table

	RegionCount	RegionTable
[23]	1345	"__aeabi_fadd"}, // 23
[24]	20	"__aeabi_fsub"}, // 24
[25]	43	"__aeabi_frsub"}, // 25
[26]	1413	"__aeabi_fmul"}, // 26
[27]	46	"__ARM_scalbnf"}, // 27
[28]	65	"__aeabi_i2f"}, // 28
[29]	205	"_float_round"}, // 29
[30]	446	"_float_epilogue"}, // 30
[31]	0	"__aeabi_fdiv"}, // 31
[32]	16	"_frnd"}, // 32
[33]	7	"__aeabi_f2iz"}, // 33
[34]	0	"__scatterload"}, // 34
[35]	53	"__aeabi_ui2f"}, // 35
[36]	111	"__ARM_common_ll_muluu"}, //
[37]	121	"__ARM_fpclassifyf"}, // 37
[38]	12	"__mathlib_flt_underflow"},
[39]	494	"__mathlib_rredf2"}, // 39
[40]	0	

Performance Impact

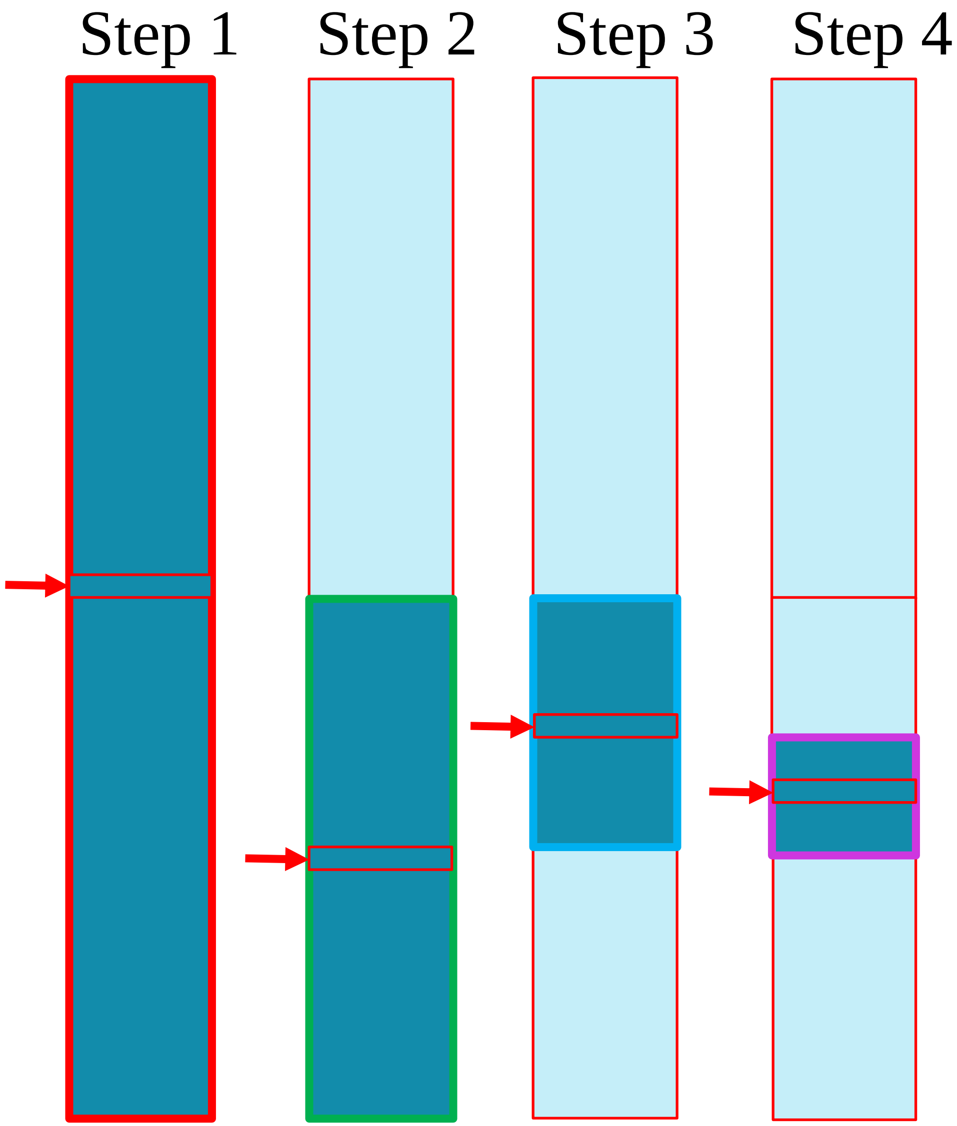
- Original table takes 55,806 comparisons
- Sorted table takes 14,934 comparisons
  - 26.8% of original value
- Profiler speed-up = 55,806/14,934 = 3.73x





Optimization 2: Binary Search

- "Divide and conquer" approach
- Requirements
  - Regions in table must be sorted by increasing starting address
  - For each entry, start address <= end address
- Start in middle of the region table
- Compare entry's start and end addresses with search address
  - If search address is within start and end addresses, then have found the region, so search is done
  - If search address is *before* start address, then repeat with *upper* portion of table
  - If search address is *after* end addresses, then repeat with *lower* portion of table
- Repeat until finding matching region or there's no table left to search



Example: Search for Address 0x380

- Start with entire table (entries 0 to 32)
- Examine middle entry of table (0 to 32)
  - Index: (32+0)/2 = 16
- 0x380 < 0x411, so repeat with upper half of this sub-table
- Examine middle entry of sub-table (0 to 15)
  - Index = (15+0)/2 = 7.5 -> 7
- 0x380 > 0x272, so repeat with lower half of this sub-table
- Examine middle entry of sub-table (7+1=8 to 15)
  - Index = (15+8)/2 = 11.5 -> 11
- 0x380 < 0x3a1, so repeat with upper half of sub-table

```
   const REGION_T RegionTable[] = {
0    {0x000000d5, 0x0000014c, "SystemInit"}, // 0
1    {0x0000014d, 0x0000022c, "SystemCoreClockUpdate"}, // 1
2    {0x00000261, 0x00000268, "Reset_Handler"}, // 2
3    {0x00000269, 0x0000026a, "NMI_Handler"}, // 3
4    {0x0000026b, 0x0000026c, "HardFault_Handler"}, // 4
5    {0x0000026d, 0x0000026e, "SVC_Handler"}, // 5
6    {0x0000026f, 0x00000270, "PendSV_Handler"}, // 6
7    {0x00000271, 0x00000272, "SysTick_Handler"}, // 7
8    {0x0000027d, 0x000002ee, "main"}, // 8
9    {0x00000305, 0x00000356, "Init_RGB_LEDs"}, // 9
10   {0x00000357, 0x0000038a, "Control_RGB_LEDs"}, // 10
11   {0x000003a1, 0x000003dc, "Init_Profiling"}, // 11
12   {0x000003dd, 0x000003e4, "Disable_Profiling"}, // 12
13   {0x000003e5, 0x000003ec, "Enable_Profiling"}, // 13
14   {0x000003ed, 0x000003ee, "Clear_Lower_Screen"}, // 14
15   {0x000003ef, 0x000003f0, "Print_Results"}, // 15
16   {0x00000411, 0x0000049a, "PIT_IRQHandler"}, // 16
17   {0x0000049b, 0x000004e6, "Init_PIT"}, // 17
18   {0x000004e7, 0x000004f2, "Start_PIT"}, // 18
19   {0x000004f3, 0x000004fe, "Stop_PIT"}, // 19
20   {0x000004ff, 0x00000550, "Init_PWM"}, // 20
21   {0x00000551, 0x0000056a, "Set_PWM_Values"}, // 21
22   {0x000005c1, 0x000005ec, "__aeabi_uidivmod"}, // 22
23   {0x000005ed, 0x0000068e, "__aeabi_fadd"}, // 23
24   {0x0000068f, 0x00000696, "__aeabi_fsub"}, // 24
25   {0x00000697, 0x0000069e, "__aeabi_frsub"}, // 25
26   {0x0000069f, 0x00000718, "__aeabi_fmul"}, // 26
27   {0x00000719, 0x00000730, "__ARM_scalbnf"}, // 27
28   {0x00000731, 0x00000746, "__aeabi_i2f"}, // 28
29   {0x00000747, 0x00000756, "_float_round"}, // 29
30   {0x00000757, 0x000007ca, "_float_epilogue"}, // 30
31   {0x000007cb, 0x00000846, "__aeabi_fdiv"}, // 31
32   {0x00000847, 0x00000884, "_frnd"}, // 32
```

Example: Searching for Address 0x380 (continued)

- Examine middle entry of sub-table (8 to 11-1=10)
  - Index = (10+8)/2 = 9
- 0x380 > 0x356, so repeat with lower half of sub-table
- Examine middle entry of sub-table (9+1=10 to 10)
  - Sub-table (10 to 10) has only one entry, so index = 10
- 0x380 >= 0x357 and 0x380 <=0x38a, so we found it! Control_RGB_LEDs was running

```
   const REGION_T RegionTable[] = {
0    {0x000000d5, 0x0000014c, "SystemInit"}, // 0
1    {0x0000014d, 0x0000022c, "SystemCoreClockUpdate"}, // 1
2    {0x00000261, 0x00000268, "Reset_Handler"}, // 2
3    {0x00000269, 0x0000026a, "NMI_Handler"}, // 3
4    {0x0000026b, 0x0000026c, "HardFault_Handler"}, // 4
5    {0x0000026d, 0x0000026e, "SVC_Handler"}, // 5
6    {0x0000026f, 0x00000270, "PendSV_Handler"}, // 6
7    {0x00000271, 0x00000272, "SysTick_Handler"}, // 7
8    {0x0000027d, 0x000002ee, "main"}, // 8
9    {0x00000305, 0x00000356, "Init_RGB_LEDs"}, // 9
10   {0x00000357, 0x0000038a, "Control_RGB_LEDs"}, // 10
11   {0x000003a1, 0x000003dc, "Init_Profiling"}, // 11
12   {0x000003dd, 0x000003e4, "Disable_Profiling"}, // 12
13   {0x000003e5, 0x000003ec, "Enable_Profiling"}, // 13
14   {0x000003ed, 0x000003ee, "Clear_Lower_Screen"}, // 14
15   {0x000003ef, 0x000003f0, "Print_Results"}, // 15
16   {0x00000411, 0x0000049a, "PIT_IRQHandler"}, // 16
17   {0x0000049b, 0x000004e6, "Init_PIT"}, // 17
18   {0x000004e7, 0x000004f2, "Start_PIT"}, // 18
19   {0x000004f3, 0x000004fe, "Stop_PIT"}, // 19
20   {0x000004ff, 0x00000550, "Init_PWM"}, // 20
21   {0x00000551, 0x0000056a, "Set_PWM_Values"}, // 21
22   {0x000005c1, 0x000005ec, "__aeabi_uidivmod"}, // 22
23   {0x000005ed, 0x0000068e, "__aeabi_fadd"}, // 23
24   {0x0000068f, 0x00000696, "__aeabi_fsub"}, // 24
25   {0x00000697, 0x0000069e, "__aeabi_frsub"}, // 25
26   {0x0000069f, 0x00000718, "__aeabi_fmul"}, // 26
27   {0x00000719, 0x00000730, "__ARM_scalbnf"}, // 27
28   {0x00000731, 0x00000746, "__aeabi_i2f"}, // 28
29   {0x00000747, 0x00000756, "_float_round"}, // 29
30   {0x00000757, 0x000007ca, "_float_epilogue"}, // 30
31   {0x000007cb, 0x00000846, "__aeabi_fdiv"}, // 31
32   {0x00000847, 0x00000884, "_frnd"}, // 32
```

Estimating the Performance Impact

linear

Bin Search

- How many iterations are needed?
  - Will take fewer iterations for regions which are located at certain entries
    - 1 iteration: entry n/2
    - 2 iterations: entries n/4, 3n/4
    - 3 iterations: entries n/8, 3n/8, 5n/8, 7n/8
    - etc.
  - Maximum number of comparisons to find an address is ceiling($\log_2$(n))
    - Ceiling(x) = smallest integer which is not smaller than x
- This example
  - 33 entries in table
  - ceiling($\log_2$(33)) = ceiling(5.044) = 6

- Speed-up over linear search
  - Linear search: on average takes n/2 iterations
  - Estimate as (n/2)/ceiling($\log_2$(n)) = 16.5/6 = 2.75x
- Speed-up increases as table size n grows
  - For 256 entry table, speed-up is 128/8 = 16x
- Extra Credit
  - Modify the profiler to use a binary search, and measure performance impact

preserved register?

f({

reg ← f

g();

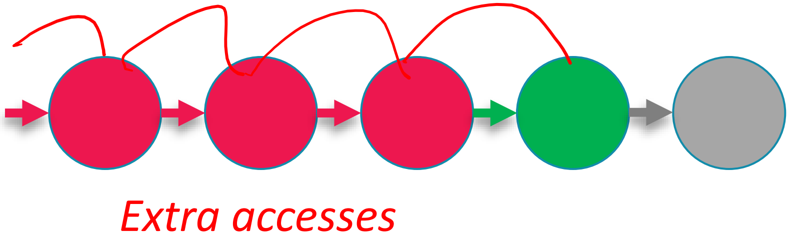
}

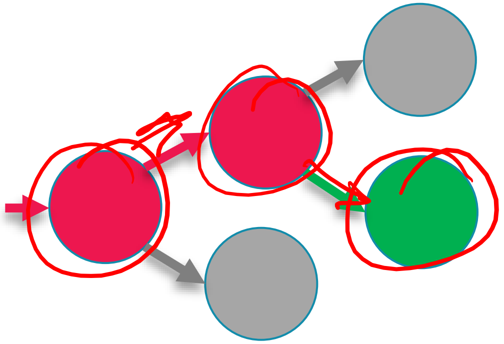
← hit

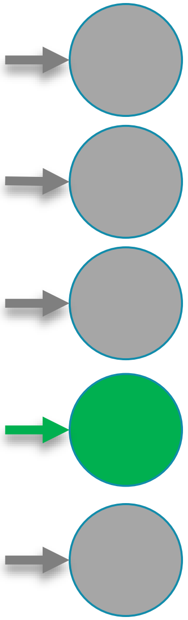
g()

reg ← g

Examples of Data Structures







- List – sequential access, linear structure
  - Each node holds a data element, may be connected to other nodes: one predecessor, one successor
  - Sequential access to data – must traverse list by visiting nodes
  - Examples: linked list, queue, circular queue, double-ended queue
- Tree – sequential access, hierarchical structure
  - Each node holds a data element, may be connected to other nodes: parent, one or more children
  - Sequential access to data – must traverse by visiting nodes, but additional connections reduce number of intermediate nodes
  - Hierarchical structure – explicit (with pointers) or implicit (with index values)
- Array – random access
  - Each node holds one element but no connection information
  - Flat structure, same time to access each element
  - *But how do we know **which element** to access?*
    - Depends on data organization and search algorithm