9Back : Making Our Plans Great Again

Maxwell Bland Leon Cheung Kilian Pfeiffer

University of California, San Diego

Abstract

The following paper describes a set of benchmarks run on the Plan 9 operating system. In particular, it uses the only actively developed branch, 9FRONT "RUN FROM ZONE!" (2018.09.09.0 Release). The goal of this project is to provide researchers and enthusiasts with greater insight into the current state of the operating system's performance and capabilities of interaction with hardware. Through tests of CPU, Memory, Network, and File System operations, we will gain insight on bottlenecks in the system's performance and the interactions between low-level (hardware) and high-level (OS) system components. These performance statistics will be contrasted with subjective experiences of "responsiveness".

1 Introduction

Plan 9 from Bell labs is a distributed operating system which emerged around the 1980's. It built upon the ideas of UNIX, but adopted an ideology that "everything is a file". Although the system was marketed in the 90's, it did not catch on, as prior operating systems had already gained enough of a foothold. Eventually, during the 2000's, Bell Labs ceased development on Plan 9, meaning official development halted. Unofficial[1] development continues on the 9front fork of the codebase, with new support for Wi-Fi, Audio, and everything anyone could ever want or need.

The experiments were performed as a group using a shared codebase and a single machine described in the following section. The measurements were performed via programs written in Plan 9's *special* version of C 99 and Alef, both described in the original Plan 9 paper todocite. The compilers used were the x86 versions of Plan 9's C compiler and Alef compiler, 8c and 8al respectively. The compilers were run with no special optimization settings. Version numbers are not available. Measurements were performed on a single machine running Plan 9 directly from hardware; given the nature of the Plan 9 system, additional metrics could be established for networked file systems and CPU servers; these measurements were not done for sake of simplicity, and because results under these conditions should be inferrable from the results cataloged within this paper.

[1]This is debatable. If you adopt an orphan, are they not your official child?

2 Machine Description

We ran this beautiful operating system of the gods on a Thinkpad T420, the machine of true developers.

```
    Processor: model, cycle time, cache sizes (L1, L2,
      instruction, data, etc.)
      Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz (Sandy Bridge)
      cache size 3072 KB
      L1$ 128 KiB
        L1I$ 64 KiB 2x32 KiB 8-way set associative
        L1D$ 64 KiB 2x32 KiB 8-way set associative
write-back
      L2$ 512 KiB 2x256 KiB 8-way set associative
write-back
      L3$ 3 MiB   2x1.5 MiB 12-way set associative
write-back
      cpu family 6
      model 42
      stepping 7
      siblings 4
      cores 2
      fpu yes
      fpu_exception yes
      fpu_exception yes
      bogomips 4986.98
      clflush size 64
      cache_alignment 64
    Details at https://en.wikichip.org/wiki/
    intel/core_i5/i5-520m

    Memory bus
      DDR3-1333
```

```
  i/o-bus-frequency: 666MHz
  bus-bandwith:  10656 MB/s
  memory-clock:  166MHz
  Column Access Strobe (CAS) latency:

I/O bus
  SataIII-speed: 600MB/s

RAM size
  8 GB
Disk: capacity, RPM, controller cache size
  Samsung SSD 860 EVO 500GB
  Capacity: 500GiB
  RPM: 550MB/s read, 520 MB/s write
   and 98,000 IOPS (Read QD32)
  Controller Cache Size: 512MB
Network card speed:
 Intel 82579 LM Gigabyte: 1Gb/s
 intel Centrino Ultimate-N 6300: 450 Mbps

Operating system (including version/release)
  9FRONT "RUN FROM ZONE!" (2018.09.09.0 Release)
```

3 Experiments

For each section, we report the base hardware performance, make a guess as to how much overhead software will add to base hardware performance, combine these two into an overall prediction of performance, and then implment and perform the measurement. In all cases, we run the experiment multiple times, and calculate the standard deviation across measurements. We use the `cycles()` syscall to record the timestamp. Dynamic CPU frequency scaling was disabled for all trials, and all trials were restricted to a single core.

3.1 Measurement Overhead (Reading Time)

The following section reports overheads of reading time. One trial involves looping over 16 `cycles` timing calls, 2^{16} times. We average out for each call of `cycles`, and we do 64 trials altogether.

Since we imagine getting the current cycles clock is a fast operation, so we estimate the hardware performance to be on the scale of nanoseconds, since one clock cycle takes approximately 1 nanoseconds. We think that the `cycles` call should do something similar to reading from the cycle counter, so it may be a bit slower, perhaps on the order of 10ns. [https://www.7-cpu.com/cpu/SandyBridge.html]

The software cost of this, however, will be tremendous, since that value must be moved out from a register and potentially to a variable, and a syscall must be made, the result might be aroudn 10 to 30 times the hardware cost.

This would put our predicted cost at around 100 to 300 nanoseconds.

	Reading (ns)	Loop (ns)
Hardware Guess	100	2
Software Multiplier	10-30	10
Prediction	1-3 usec	20
Average	11	17.2
Std Dev.	0	0

Figure 1: Measurement Overheads

3.2 Measurement Overhead (Loop Overhead)

In order to measure the loop overhead, we took the time at the end of a loop iteration and at the start of a loop iteration, and find the difference between those to measure the loop overhead time. We repreat this 16384 times within each trial, and perform 64 trials. We average over all of these trials.

The cost of doing a single for loop based branch in hardware is also very low. Assuming a compare, jump, and register increment takes 3 cycles, and adding 2 cycles for rare branch mispredictions, the loop overhead should be around 5 cycles. Adding software uncertainties, the loop overhead could be about 50 nanoseconds. [https://www.7-cpu.com/cpu/SandyBridge.html]

Software, again, will slow this down, as well as the rest of the operating system. Other system processes may need the processor, data may not be in the carch, and thus the latency might be 20 times this, the same as a reference to the L2 cache, roughly.

Thus, our predicted cost is around 2 microseconds.

3.3 Procedure Call Overhead

In measuring the function overhead, we created functions of zero to seven integer arguments, we then proceeded to call eacah of these functions 20000 times, and found the average time taken over these 20000 calls. Overall, increment overhead was almost non-existent.

The hardware cost of a procedure call using the standard c calling convention should not be that high, as it is almost all done in registers. The caller pushes the arguments to the stack (very little cost), saves the return address, the frame pointer, and calls the function. The hardware cost of this should be more than a for loop, but still reasonable, maybe roughly equitable since there are no comparisons or complex operations. Thus, somewhere on the range of 150ns to 200ns would seem reasonable, maybe even 100ns. [https://www.7-cpu.com/cpu/SandyBridge.html]

The software cost of this is is probably about the same as the measurement overhead and for loop overhead, since we still may need to context switch to another process, as well as handle interrupts, etc. So I would reason that it would be around 20x the hardware cost.

This puts our prediction for the cost to be somewhere between 2000 nsec and 4000 nsec. Arguments shouldn't really

Hardware Guess	150 nsec	
Software Multiplier Guess	20x	
Prediction	3000 usec	
Num Args	Average	Std Dev.
0	2.110 usec	0.191 usec
1	2.104 usec	1.248 usec
2	2.118 usec	0.346 usec
3	2.113 usec	0.571 usec
4	2.108 usec	0.213 usec
5	2.120 usec	0.915 usec
6	2.111 usec	2.165 usec
7	2.108 usec	0.321 usec

Figure 2: Procedure Call Overheads

	System Call Overhead
Hardware Guess	100 nsec
Software Multiplier	10x
Prediction	1 usec
Average	0.866 usec
Std Dev.	0.0144 usec

Figure 3: Syscall Overhead

effect the speed of a function call under our system.

3.4 System Call Overhead

To measure the syscall overhead, we used the `errstr(2)` syscall, in particular, `rerrstr(char*, vlong)`. `reerrstr` reads the error string for the error of a previous syscall, and does not clear the errstr buffer. Within each trial, we performed the syscall 16384 times, and averaged over 64 trials.

Data from IBM puts the cost of making a system call on the Pentium processor in 1997/1995 to be 223 cycles, or roughly 1.68 usec for that processor. Our processor is much faster, putting the timing at around 100 nsec.

It is likely that the overhead of a syscall beyond this is pretty high due to the actual operations and implementation of the syscall operation. Thus, the cost could be 30 to 40x the hardware cost.

That puts our estimate at around 3 to 4 usec.

3.5 Task Creation Time

We measured process creation overhead in two different ways, one in which we `rfork`'ed a new process without copying the parent state to the child, and, `fork`, where the child will copy all of the parent's file descriptors, and share the other state as normal. We do these 20000 times for each type of `fork` and we average over all of these calls. This methodology results from the fact that Plan 9 does not have kernel threads.

	Light Fork	Heavy Fork
Hardware Guess	500 usec	500 usec
Software Multiplier	2x	2x
Prediction	1000 usec	1000 usec
Average	27.250 usec	28.674 usec
Std Dev	1.83 usec	58.068 usec

Figure 4: Fork Overheads

The hardware costs of a light fork should be much less than a heavy fork, since resources do not need to be shared, but because our processes are minimal, it is likely this difference is negligible. Still, this whole process will take tens of thousands to millions of cycles, meaning thousands of microseconds, maybe 500 usec with a 1,000,000 cycles. [http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/]

The software costs are most likely pretty low since Plan 9's rfork mechanism is relatively light weight and similar to the Unix version of fork. It might only scale this already tremendous runtime by a factor of two or so.

That puts our estimate at around 1000 usec for both fork operations.

3.6 Context Switch Time

Finally, to measure context switching, we created a pipe in a parent process, and forked off a new child process. We forced a context switch by writing to the pipe in the parent and then waiting for the child to read from the pipe and terminate. We take the time between when the parent goes to sleep and when the parent wakes up again after the child's termination. Then, we subtract the overhead of reading from the pipe and closing the pipe (which happens in the child), and divide the time in half to account for the two context switches. We repeat this measurement 1000 times and use a new pipe each time. To take the time of a context switch, we take the minimum time over all of these trials.

We estimated the base hardware cost of context switching while pinned to a single CPU to be on the order of around 1100 ns based upon benchmarks done on the Intel E5-2620, another Sandy Bridge processor model. [https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html]

The software cost to a context switch for Plan 9 is going to be huge. Probably on the order of 40x or something along these lines, since the OS needs to deal with scheduling and piping and I/O and virtual memory, etc..

This puts our prediction right around 44 usec.

3.7 RAM Access Time

To measure ram access time we allocate a 1.6 GB array on the heap, and iterated through it with changing stride size to

prediction L1 latency	2-3 cycles	0.8-1.2 nsec
prediction L2 latency	10 cycles	4 nsec
prediction memory latency	100 cycles	40 nsec

Figure 5: RAM access time prediction

L1 latency	6 nsec
L2 latency	10 nsec
Memory	250 nsec

Figure 6: RAM access time measurements

hit L1, L2 cache and finally memory. The stride size gets increased by factor 1.01 for each iteration. Predictions are presented in fig. 5, while the measurements are presented in fig. 6. Figure 7 presents the plot of the mean in orange and standard deviation in blue with the last index accessed in the iteration on the x axis, and memory access latency is displayed on a logarithmic scale y axis. The measurements are not completely free of overhead, but still a transition from L1 to L2, and from L2 to memory is visible in the graph.

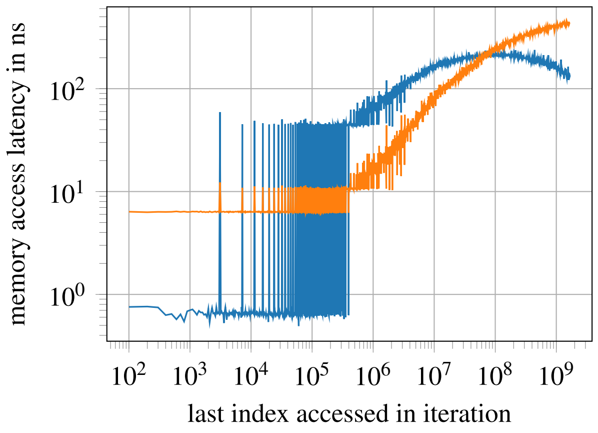

Figure 7: RAM access time with size of the memory region accessed on x axis in an logarithmic scale and logarithmic latency in seconds on the y axis. The orange graph displays the mean latency, while the blue graph displays the standard deviation

3.8 RAM Bandwidth

We measured RAM bandwidth by allocating an array of $5 \cdot 10^6$ 4-byte integers, and iterated through it to measure read and write bandwidth. For the write bandwidth, we use `memset` to write to the entire array. In order to measure the read bandwidth, we looped through the entire array, referencing each integer element one by one, and compensated for the loop overhead after averaging over all the times.

Our memory bus bandwidth is about 10GB/s, so this serves as our predicted bandwidth for both reading and writing. We don't expect too much software overhead because we can directly read from and write to memory, provided the pages are faulted in for us.

3.9 Page Fault Service Time

We create an array of size greater than the physical RAM memory in order to force the memory management system to start swapping to disk. We then iterate through this array by jumping in increments of 20 pages.

We expect faulting in from disk to be on the order of milliseconds, since our filesystem is installed on a SSD.

3.10 Discussion

Compare the measured performance with the predicted performance. If they are wildly different, speculate on reasons why. What may be contributing to the overhead? Our hardware estimates were very, very rough. It'd be worthwhile to

Hardware Cost	1100 nsec
Software Multiplier	40x
Prediction	44 usec
Average	13.458 usec
Std Dev.	9.569 usec
Min	7.862 usec

Figure 8: Context Switch Overheads

	Bandwidth (GiB/s)	
Theoretical Maximum r/w	10	
Software Multiplier	0.8	
Prediction	8	
	Average	Std Dev.
Write	6.879842	1.9829
Read[2]	9.109494	1.2845

Figure 9: Memory Bandwidth

	Page fault time (ns)	
Hardware	10^6	
Software Multiplier	1.5	
Prediction	$1.5 * 10^6$	
	Average	Std Dev.
Page fault	TODO	TODO

Figure 10: Memory Bandwidth

[2]This measurement was done using optimization, without it the read speed is 1.17

go back and double check these and discuss with our classmates on the numbers we arrived at.

Evaluate the success of your methodology. How accurate do you think your results are? Our results agree with the Plan 9 paper, which is suprising given that our processor is a bit faster. It will be worth it to discuss with Geoff our results and what we might try looking into going forward. It was hard to ensure we were taking the best approach. We certainly measured roughly what we wanted to measure.

Answer any questions specifically mentioned with the operation. The issue here is that there are no kernel threads in Plan 9, so some of the CPU measurement questions did not apply

4 Conclusion

We should try to correct our idea of hardware cycle counts for typical instructions and discuss the manners in which we chose to benchmark in order to make sure they are the absolute best way to measure these values, or something close to it.

References