Sea2Cloud, Inc.
Created February 24, 2022 04:33 PM

Modified March 7, 2022 09:05 PM

Measuring System Performance

Using Cryptographic Hashes and 'dd'

by

Robert E. Novak

Software Architect

(retired)

CONTENTS

1. INTRODUCTION

1.1 APPALLED AT BOGOMIPS

I enrolled in a course offered by the Linux Foundation and whiFle preparing to begin the course, I followed their instructions to run a script called “ready-for” to insure that my system was capable of executing the courseware and the example programs and scripts in the course. I was appalled to discover that my system was NOT compliant with their requirements. They were using Bogomips as a metric for system CPU performance. Their suggested performance level was 2,000 Bogomips and my system was running a measly 108 as its measured by Bogomips. The reason I was appalled was that I had been successfully running Ubuntu on my Raspberry PI 4 machines for several months. Granted, they were slower than my Dell Optiplex980 deskside machine, but not 20 times slower. This triggered writing some scripts to perform basic system performance and attempt to get a mixture of CPU and I/O performance measured at the same time[1] vs. the CPU-only metric of Bogomips.

In the process of running these tests, I ran into a number of surprises when looking at the relative performance of the various cryptographic hashes.

1.2 PROBLEMS WITH BOGOMIPS

Bogomips is a simple busy loop run during system boot to get a thumbnail read on CPU performance. As a result of using Bogomips[2] as a proxy for performance you run into two problems:

1) It does not take into account changes in CPU architecture that have taken place. Many CPUs (e.g. ARM) use frequency scaling to reduce the amount of power consumed by the CPU, but winds up skewing the results of the busy-wait loop shown in Bogomips.
2) It is only a measure of CPU performance and does not take into account the operational speed of the operating system and the I/O subsystems (and peripheral devices) that read/write files or perform network I/O.

1.3 AN ALTERNATIVE APPROACH

As an alternative approach I developed a set of scripts that use cryptographic hashes as a function that uses both reading of data (only modest writing) and computation intensity to compute a cryptographic hash of the input data. My first pass at this I simply ran many iterations of computing a cryptographic hash of the default dictionary in the system. Because the dictionary will vary by locales (i.e. different dictionaries for different languages), I normalized the results by computing a rate of the number of megabytes/second that were processed by the iteration over the dictionary to yield a rate in megabytes/second for the cryptographic hashing.

[1] I had been the founding chairman of the SPEC Steering committee in 1990 and 1991 when we developed performance testing to specifically measure the performances of a set of CPUs from competing companies including Intel, AMD, Mips, Sun et. al.

[2] https://en.wikipedia.org/wiki/BogoMips#Proper_BogoMips_ratings

2. Using a Raspberry PI

2.1 RASPBERRY PI OS VS. UBUNTU

I have been using Raspberry PI 3 computers for several years now and I jumped at the opportunity to purchase a Raspberry PI 4 computer when it first came out, even though it only had 4GB of RAM. A later version[3] was released with 8GB of RAM.

When I started using the Raspberry PI 3 computers I found that they were sluggish compared to commercial desktops and laptops from Dell and I would constantly run into a system stopping when I opened too many browser windows or ran too many simultaneous applications on the machines, but remember that at this point in time, circa 2019, that I felt that this was not bad considering that these computers were only $35. They were fine for running basic tasks. Not only was the Raspbian operating system free, but so is LibreOffice, so that for home use, I had a complete set of tools for running multiple computers and applications on them.

It was only this year, in 2022 that the 64-bit of Raspberry PI OS was released. Up until this point, only experimental versions of 64-bit operating systems were really available for the Raspberry PI. Previously the 4 core ARM cpus had been running 32-bit versions of the operating system, utilizing only a subset of the power of the cpus.

2.2 THE IMPACT OF ZRAM

I had realized that the reason for the sudden halting of my computers with too many browser tabs open or too many applications running was that the system was running out of memory. Increasing the amount of “swap” space that is used on the file system lead to a different long term problem with using my Raspberry PI. The microSD flash drive is used to hold the operating system and the file system for the Linux operating system from which the Raspbian operating system was derived.

One of the consequences of leaving swap on the flash drive is that it will rewrite data so often, that is causes the flash drive to fail. This is due to the heavy read/write activity on the swap file as programs were swapped out of ram to the flash drive. This causes the number of rewrites to a given chunk of the flash drive (limit of about 1000 rewrites per “cell”) to wear out the flash drive. Then the system will no longer be able to write data to “swap”. This will make the system halt.

I discovered this when I attempted to re-flash the operating system on the microSD drive and even if it would boot, it would quickly crash until I flashed the operating system on (at that time) a new $7.00 32GB microSD drive. That would run happily for a few weeks and then crash again.

At times the system was slow and required patience while you waited for applications to swap out of memory and other applications to swap in, but this was a $35 computer, not a $350+ laptop.

[3] thanks to Micron for making the memory chip — a personal plea for 16GB, since, as these tests illustrate, memory matters almost more than CPU speed

Eventually I stumbled across zram[4]. I found it on github at:

https://github.com/mdomlop/systemd-zram

I cloned it and made minor tweaks because I had systems with only 2GB of space and I felt I needed additional swap space beyond what was in this repository and I placed that on:

https://github.com/sailnfool/systemd-zram

I had a great deal of success using this since suddenly instead of reading/writing blocks of data to my microSD drive, the CPU was compressing the in-memory data and writing it to a set-aside portion of ram that appeared to be a block device. Since these are memory to memory operations, the performance was noticably faster than my prior experience. As an added bonus, I never ran into the problem of the microSD flash drive failing and needing to be replaced with a new drive.

I still had to be patient. Although The Raspbian OS is magnificent (it is a full blown Debian Linux port to an inexpensive machines), having different OS on different machines was cumbersome... I would have to "switch contexts" as I moved from machine to machine. The rest of the machines in my home were Ubuntu (save for one Windows machine because some applications and websites just don't run on Linux yet). There were experimental 64 bit server (not desktop) versions of Ubuntu that were available and despite my ancient experience with command line only UNIX[5], I prefer having a modern X-Window desktop that lets me use both command line and GUI interfaces.

2.3 WHY UBUNTU?

Finally in late 2021 the 64 bit versions of Ubuntu were available for the Raspberry PI. I was thrilled to find this available since now I would be running the same operating system on both my Raspberry Pi machines, my desktop machine[6] and my laptop[7]. I thought I would be able to build and run all of my software across all of my machines using the same APIs and libraries without a difference between them. As I discovered the hard way, this was NOT the case. I additionally made my task slightly more complicated by choosing to also perform testing on a Raspberry PI 3 with the new (in 2022) 64-bit Raspberry PI OS.

2.3.1 Zram is not installed by default on the Raspberry PI

The first thing I ran into was that zram was no longer working on Ubuntu. It took me several months to track down the reason and the fix (now incorporated into my system-zram[8]

[4] https://en.wikipedia.org/wiki/Zram
[5] circa 1974 on a PDP-11/45 which was node #6 on the Arpanet in the basement of the Digital Computer Lab, now the Donald B. Gillies lab in Urbana, Illinois on the campus of the University of Illinois Urbana-Champaign where I was an undergraduate student and thanks to Prof. Gillies (RIP) I had access to that machine.
[6] Dell Optiplex980
[7] Dell Inspiron3185
[8] https://github.com/sailnfool/systemd-zram

repository). Contrary to the notes in kernel documentation[9] the zram module is NOT automatically provided in all kernels. With the Ubuntu 21.10 release, Ubuntu removed the zram module and it must be installed as system add-on with "apt" or "apt-get." The resolution is described in a github repository[10]. I can't recall where, but I had read that the reason was to reduce the size of ramfs for allowing Ubuntu to run on machines with a small memory footprint[11].

2.3.2 Cryptographic Hash functions missing on Raspberry PI

The second thing I ran into was the missing cryptographic hash functions. Commands like sha1sum, sha256sum and sha512sum were missing from the default Ubuntu 21.10. To fix this you need to:

```
sudo apt install coreutils
```

2.3.3 Timing functions missing

The third thing I ran into was the missing "timing" command /usr/bin/time. This problem did not occur on Ubuntu, but on 64-bit Raspberry PI OS. I needed this to measure the amount of time required for test applications. To fix this you need to:

```
sudo apt install time
```

I am certain there are many other differences, but that is not the point of this document.

3. Using Cryptographic Hashes for Performance Testing

I chose to use the available cryptographic hashes since they are widely available in the GNU coreutils library which are available for most distributions. Because these tools are based on the cryptographic libraries for GNU and Linux distributions, the focus on cryptocurrencies has hopefully meant that these libraries are well tuned for best performance. I chose to use four different cryptographic hash functions that are well known.

3.1 THE BLAKE2 HASH (B2SUM)

The Blake2 hash was a contender for the SHA3 NIST standard for cryptographic hashing. Apparently the committee chose the SHA256 algorithm because it had an advantage in hardware

[9] https://www.kernel.org/doc/Documentation/blockdev/zram.txt
[10] https://github.com/ecdye/zram-config/issues/71
[11] I can only speculate that they are trying to fit into devices with only 1 GB of RAM since I have had no memory issues on Raspberry PI 4s with 4GB and 8GB of RAM. Perhaps to address the IoT market?

implementations[12]. The software is FOSS and has been freely available for use since 2012. According to the website: https://www.blake2.net/ :

> *BLAKE2 is a cryptographic hash function **faster than MD5, SHA-1, SHA-2, and SHA-3**, yet is at least as secure as the latest standard SHA-3. BLAKE2 has been adopted by many projects due to its high speed, security, and simplicity.*
>
> *BLAKE2 is specified in RFC 7693, and our code and test vectors are available on GitHub, licensed under CC0 (public domain-like). BLAKE2 is also described in the 2015 book The Hash Function BLAKE.*

The results that I generated below did not contradict this in any way.

The Blake2 cryptographic hash that is generated by b2sum is a 256 bit hash code that is a unique number that identifies the source code that is fed to the algorithm. You can find an introductory text to the Blake function in Wikipedia at:

https://en.wikipedia.org/wiki/BLAKE_%28hash_function%29

but the full text at blake2.net is probably more worthwhile.

3.2 THE SHA-1 CRYPTOGRAPHIC FUNCTIONS

The SHA-1 function is widely used since 1995, especially in git, but since 2005 it has been considered cryptographically broken and has been deprecated in most applications. It is included here to show that although it is fast, and small (only an 80 bit hash), which is the apparent reason for choosing it for git[13], for the git function which is more pragmatic than requiring it to be secure in order to detect differences in repository elements. It may no longer be the best choice for git[14], but it is unlikely that this change will ever take place.

3.3 THE SHA256 CRYPTOGRAPHIC HASH

This cryptographic hash is part of the SHA-2 family of hash functions and was patented on Dec. 7, 2004, although the patent has been released by the US Government as royalty-free to encourage use of the cryptographic method. The algorithm shows its age in the results.

[12]With the falling cost of CPUs driven by ARM and RISC V Cpus, it is not clear how large the market will be for pure hardware cryptographic devices versus systems with inexpensive CPUs and software implementations.

[13]There may not have been robust additional choices available at the time that git was first written.

[14] My two cents worth here is that whenever a cryptographic hash is used there should be two to four bytes as the prefix to the hash which canonically identifies with a number the identity of the hash that follows. This would allow over time that hashes that have become obsolete (as md5 and sha-1 have) to identify their use and when source documents are available to convert obsolete hashes to newer hashes. All of this may be moot when quantum computing arrives, but I believe that for pragmatic applications existing numerical cryptography solutions will be in use for a long time.

3.4 THE SHA512 CRYPTOGRAPHIC HASH

This cryptographic hash is also part of the SHA-2 family of hash functions. Although the algorithm is fundamentally the same as the other SHA-3 functions, the wider internal tables and resulting 512 bit hash code make this very robust with regard to existing von Neuman architecture CPUs.

4. SYSTEMS UNDER TEST

The Table 1: Systems under Test outlines the characteristics of the systems that were used for testing

Table 1: Systems under Test

System Name	Model	CPU Architecture	Cores	CPU Max MHz	CPU Mfg	RAM GB	Vintage	Purchased
opti	Optiplex980	x86_64	4	3193	AMD	8	2010	Refurb
inspiron	Inspiron4185	x86_64	2	1800	AMD	8	2018	Refurb
Hplap		x86_64	4	2000	AMD	6	2015	New
PI04-04-02	Raspberry PI 4	aarm64	4	1500	BCM	4	2019	New
PI04-08-03	Raspberry PI 4	aarm64	4	1500	BCM	8	2020	New
pi3	Raspberry PI 3	aarm64	4	1200	BCM	1	2019	New

5. TESTING METHODOLOGY

5.1 OVERALL METHODOLOGY

The overall methodology is to create a small script in /tmp which will run the test over a number of iterations. The amount of elapsed time, system time and user time is measured with:

```
/usr/bin/time
```

and after the script is run, the timing data which was saved in a file is parsed to collect the timing information. For each test instance of a cryptographic hash, a number of copies of the dictionary and the number of iterations, a test result line is produced with "|" characters separating the fields.

In order to minimize I/O effects of the testing (I was primarily interested in CPU performance), I sent the output of the cryptographic hash to:

```
/dev/null
```

... the proverbial “bit-bucket” disposal of output so that the output is not sent to a physical device.

I have three different test scripts that I generated each time using a bash “here” script for each of the test scripts each time the test is run (the creation and selection of these tests are not part of the testing time).

5.1.1 Cryptographic Hash with output to “/dev/null”

The first script, illustrated below invokes the selected cryptographic hash for the number of copies of the dictionary for the chosen number of iterations and sends the output to /dev/null.

```
cat > ${TIMER_APP_DEV_NULL} <<EOFNULL
#!/bin/bash
i=0
while [[ \${i} -lt ${iterations} ]]
do
  ${hashprogram} ${TIMER_INPUT} >> ${OUTFILE}
  i=\$((i+1))
done
exit 0
EOFNULL
```

The TIMER_INPUT is a hard link to the file which contains the requisite number of copies of the local dictionary. For those that want to re-run these tests in their locale, it is easy to manualy change the default dictionary used by the scripts. Since the scripts normalize the data to a rate in MB/Sec, the exact input file is irrelevant. Note that since the copies of the diectionary (whether 1 copy or 512 copies) are made in /tmp, the actual copy is made only one time.

5.1.2 Cryptographic Hash with output to a file

The second script, illustrated below invokes the selected cryptographic hash the number of copies of the dictionary for the chosen number of iterations and sends the output to a file in “/tmp.” I chose not to use this script. The output write speed was not important for the purposes of hese tests, but might be important for a different set of tests.

```
cat > ${TIMER_APP} <<EOF
#!/bin/bash
rm -f ${OUTFILE}
i=0
while [[ \${i} -lt ${iterations} ]]
do
  ${hashprogram} ${TIMER_INPUT} >> ${OUTFILE}
  i=\$((i+1))
done
rm -f ${OUTFILE}
exit 0
EOF
```

5.1.3 "dd" command with output to "/dev/null"

The third script, illustrated below invokes the "dd" command on a number of copies of the dictionary for the chosen number of iterations and sends the output to /dev/null. There is a fuller description of why this was chosen under the Method description.

```
cat > ${TIMER_APP_DD} <<EOFDD
#!/bin/bash
i=0
while [[ \${i} -lt ${iterations} ]]
do
  ${hashprogram} status=none if=${TIMER_INPUT} of=${OUTFILE}
  i=\$((i+1))
done
exit 0
EOFDD
```

Note that this script is different due to idiosyncracies of the dd command. To suppress output from the "dd" command I need to provide the parameter "status=none." rather than redirecting the stderr output to /dev/null.

5.2 METHOD 1 SINGLE COPY OF THE DICTIONARY WITH MULTIPLE ITERATIONS

I chose to run the basic test in two ways. The first method was to test each of the cryptograhic hash sum programs by running multiple iterations of the hash of the default dictionary on Ubuntu:

/usr/share/dict/american-english

This dictionary was copied into:

/tmp/american-english_1

This name was then linked to the filename that would be used for the test script (TIMER_INPUT)

This testing was performed over multiple iteration counts for 1Kibibyte to 50 Kibibyte for both the cryptographic hashes and the "dd" copy operations.

5.3 METHOD 2 - 512 COPIES OF THE DICTIONARY WITH MULTIPLE ITERATIONS

This test was run in the same two ways. Under both this method and the single copy method N (1 or 512) copies of the dictionary are copied into:

/tmp/american-english_512

This name was then linked to the filename that would be used for the test script. Although not perfect, I observed that when this copy was done, that a subsequent test of the cryptographic hash or the "dd" command would have lower than anticipated results. The reason for this is that Linux performs "write behind" operations. Even though a program has completed

the write operations, the actual writes are buffered in memory until the physical media has time to complete all of the write operations. To minimize this effect, the first time I create a copy, I take the number of copies (in this case 512) and divide that by a number (in this case 8[15]) and have the system "sleep" for 512/8 => 64 seconds to allow the I/O operations to complete. Note that this is ONLY done the first time that this number of copies are used on a given computer. My observation was that without the sleep the hash rate was approximately 50% of the hash rate on subsequent runs. With the sleep command included, the first run is slower by 1 part in 108 MBps or 99.074% of the later runs. Subsequent tests are unaffected.

Note that when the testing against multiple copies of the dictionary, on each system under test, I created the copy in /tmp so that until the system is re-booted, the number of copies persists and the copy/sleep operation is only performed once after a system is rebooted.

Note that in the scripts that run the tests, the number of copies and the sleep time divisor are all parameters to the scripts to allow flexibility in testing in your environment. The code for creating the number of copies and sleeping is shown below.

```
#####################################################
# In order to minimize the overhead for each test, if the copies of the
# dictionary have already been done, then just link the input file to
# that number of copies.  Otherwise we make the number of copies of the
# dictionary in /tmp and link that to the TIMER_INPUT
#####################################################
if [[ -r /tmp/${language}_${numcopies} ]]
then
  ln /tmp/${language}_${numcopies} ${TIMER_INPUT}
else
  for i in $(seq ${numcopies})
  do
    cat ${dictpath}/${language} >> /tmp/${language}_${numcopies}
  done
  echo "Sleeping for $((numcopies/waitdivisor)) seconds to allow copy to
complete"
  sleep $((numcopies/waitdivisor)) # Observed behavior is that the first time the
file is
        # created it impacts the timing of the program under test
        # for large files, presumably waiting for this write
        # operation to finish. The value of 5 seconds is arbitrary
        # further testing should be done to make this wait time be
        # a function of the numcopies.  The formula just added
        # assumes 5 seconds for 100 copies and scales from there
  ln /tmp/${language}_${numcopies} ${TIMER_INPUT}
fi
```

[15] This was observed by performing a "cat" operation to append 512 copies of the dictionary to a temporary file. On my systems I have running graph illustrating the Read/Write I/O rate and I experimented with different divisors on my Raspberry PI computers to insure that the sleep time was sufficient to allow the system to quiesce following the copying. The graphs in question, come from a Gnome extension called "system-monitor" by Cerin (see https://extensions.gnome.org/#) and search for system monitor. I leave these running ALL of the time on my Ubuntu systems to show me CPU, Memory, Swap, Network I/O and disk I/O. To minimize effects I have relatively long delays (2-5 seconds) between updates, but when the system is misbehaving a quick glance at these graphs can give me insight into what is happening.

5.4 METHOD 3 USING "DD" AGAINST THE COPIES OF THE DICTIONARY

As I noted above, in looking at the rate of hashing (measured in MB/second) for the iteration over single copies of the dictionary vs. multiple copies (e.g., 512) of the dictionary, I observed that the hashing rate was significantly slower. I chose to use the "dd" command to read and copy the same copies of the dictionaries to "/dev/null" in order to look at subtracting the "dd" time from the hash time to simulate a "pure" hash time number without the overhead of the open and read operations by the cryptographic hash sum programs. This will be discussed in results.

6. RESULTS

In all of the "rate" tables that follow, I have scaled all of the graphs to the same MB/Sec scale to allow easy visual comparison of the tables.

6.1 B2SUM

The first test was b2sum, and the slow initial rates prompted me to test with not just one copy of the dictionary but also with 512 copies of the dictionary in one file and then iterate from 1024 to 15Kib iterations.

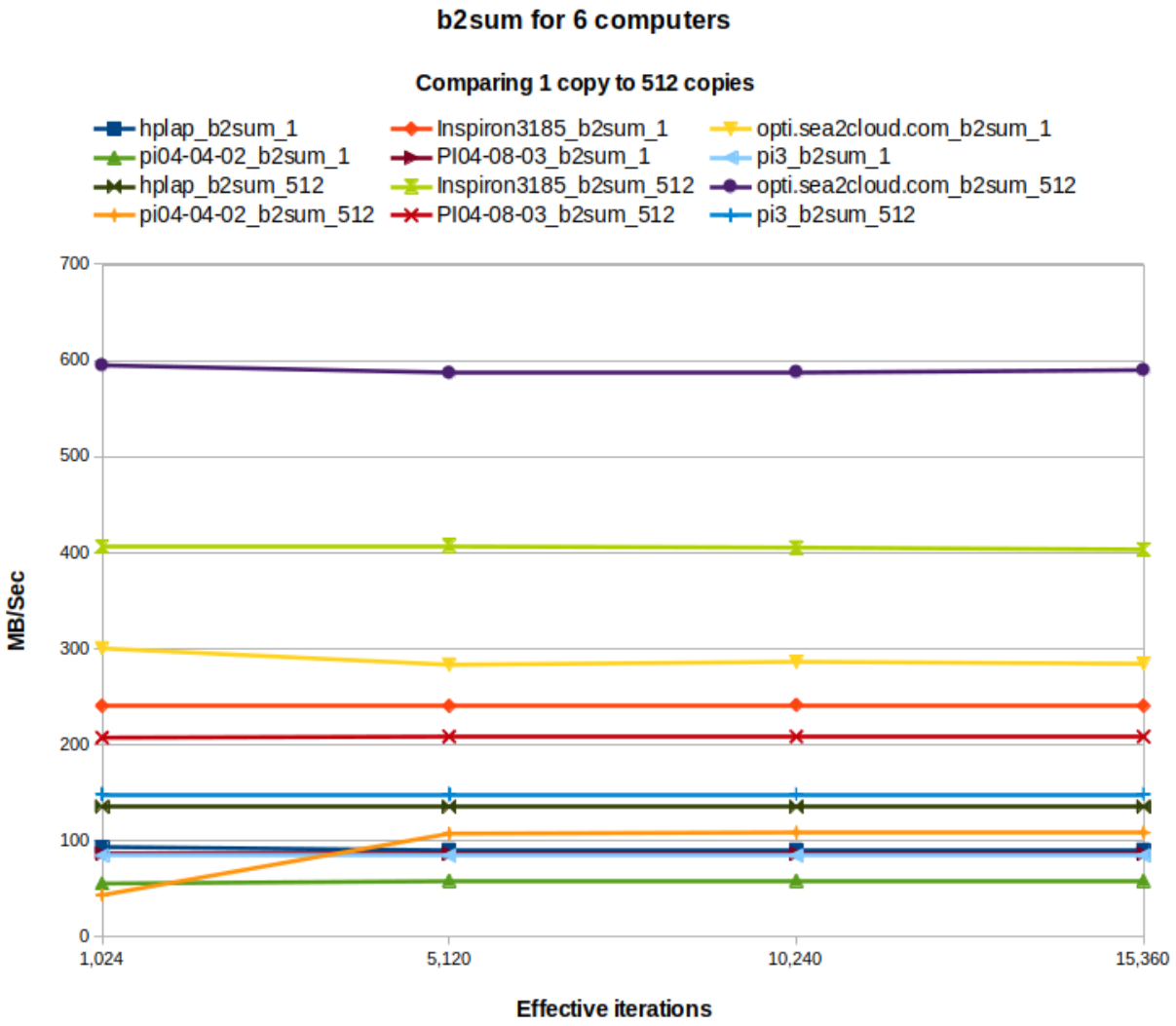

Figure 1: b2Sum Results

Note that on my deskside computer ("opti") that the hashing rate reaches nearly 600 MB/sec on 512 copies of the dictionary, about double the ~300 MB/Sec of iterations of the single copy of the dictionary. My theory for the difference is that the actual I/O of the file is overwhelmed by the I/O that system has to perform in loading the application binary into memory so many times. A "hidden" part of the Effective Iterations in each graph is that in order to hold the amount of data processed identical, when processing the 512 copies of the dictionary, I divide the number of iterations by 512. The same amount of data is read. For example for 1024 Effective iterations of the dictionary, the application is actually run 1024 times, but for 512 copies of the data, the application is only run twice.

6.2 SHA1SUM

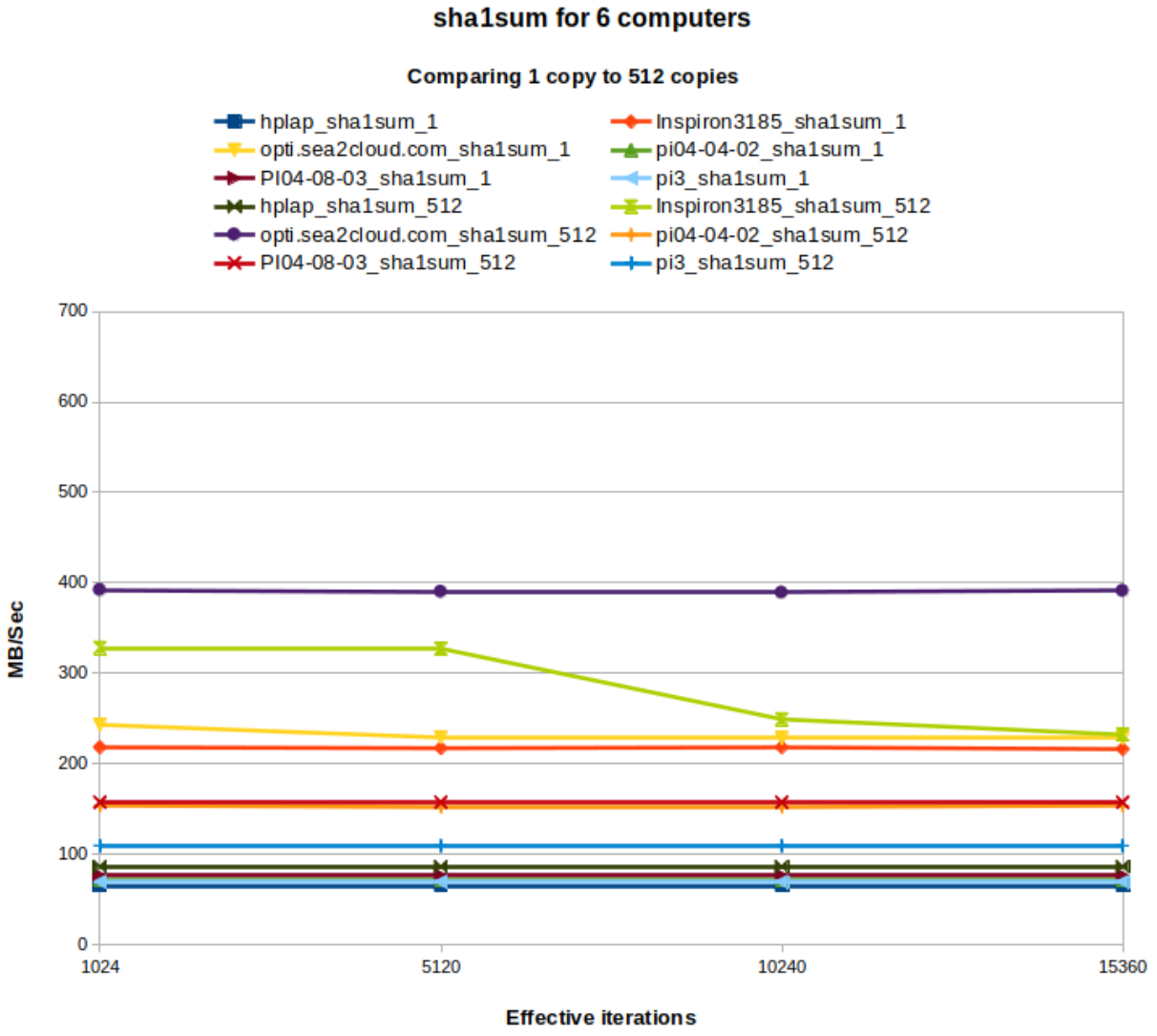

Figure 2: sha1sum results

Note that sha1sum is about 2/3 of the maximum speed of b2sum, although the generated cryptographic hash is smaller, the processing time is significantly greater. Other comparisons of the 256 bit hash generated by b2sum, vs, the 80 bit hash generated by sha1sum could argue for using the smaller hash code for both speed and storage minimiation despite the cryptographic deprecation of sha1. As time moves forward, both the storage space and CPU considerations become less, since more efficient algorithms exist as time moves forward and storage space becomes less of a consideration as both the cost per bit of purchasing storage continues to fall and throughput rates for memory and peripherals increase.

6.3 SHA256SUM

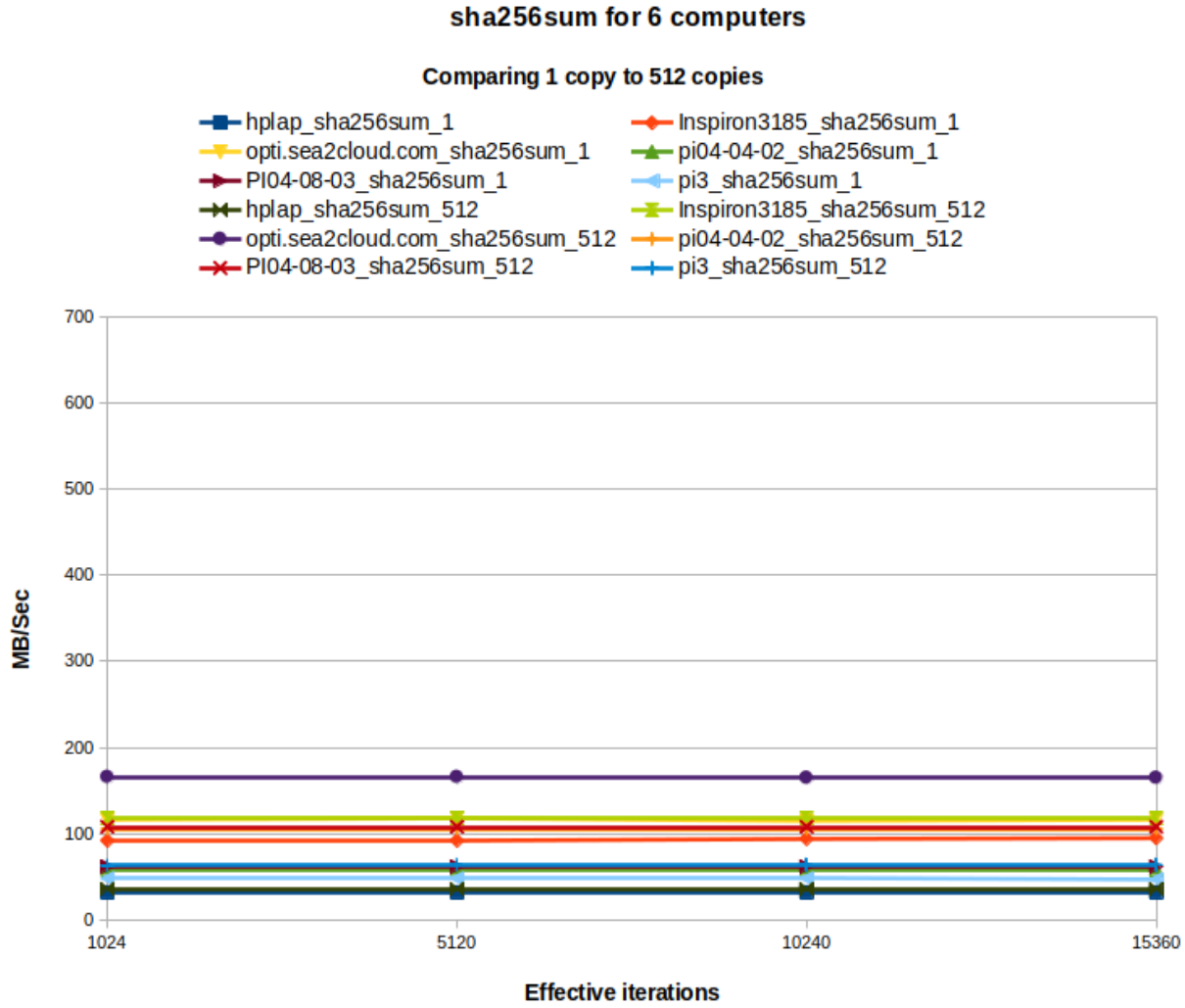

Figure 3: sha256sum results

This Figure 3: sha256sum results clearly shows that the sh256sum is the worst performer of all of the cryptographic hashes tested.

6.4 SHA512SUM

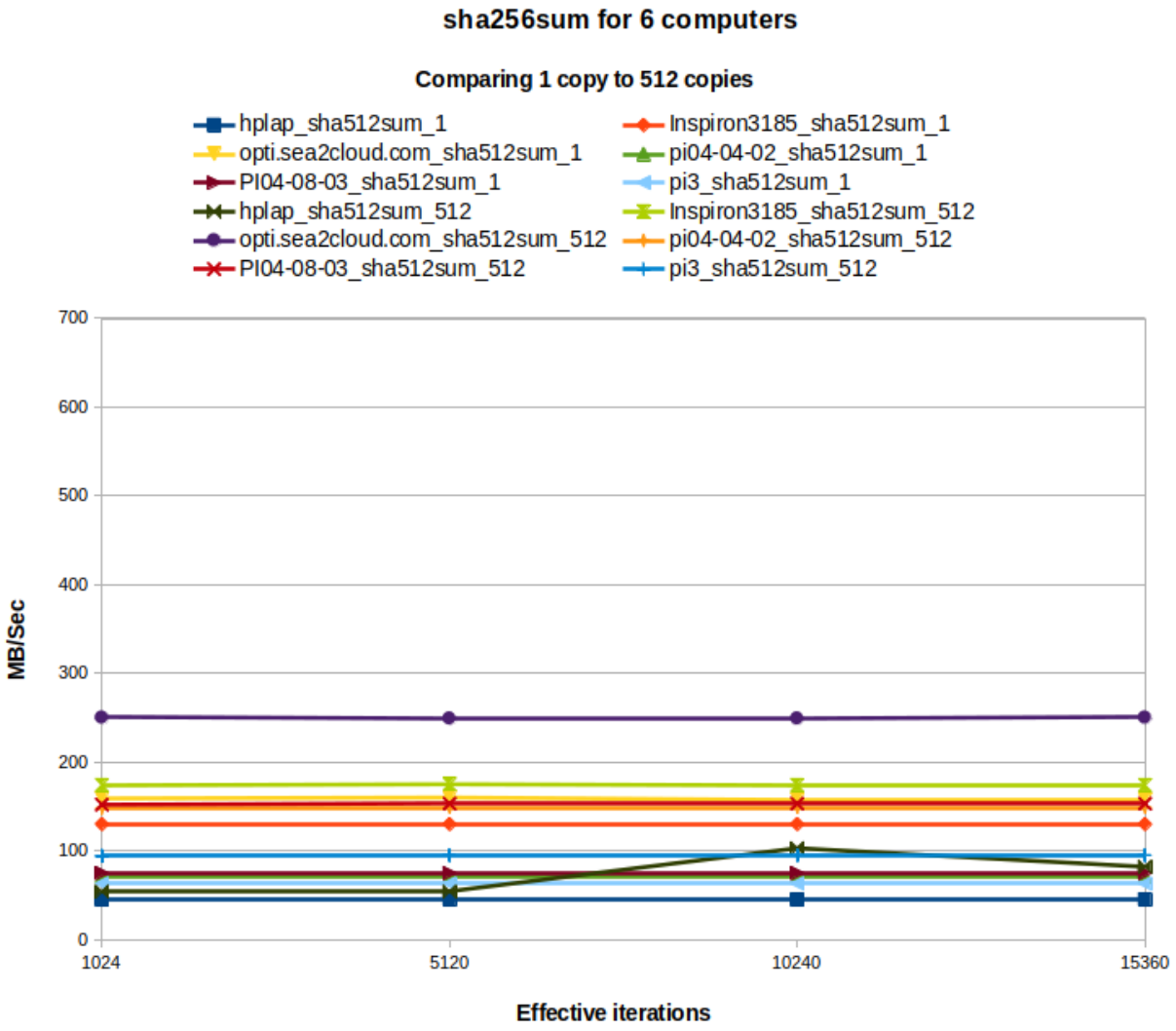

Figure 4: sha512 results

This Figure 4: sha512 results clearly shows that sha512sum outperforms the sha256sum algorithm, even though it generates a larger 512 bit cryptographic hash. However it is still anemix when compared agains Blake2's b2sum performance.

6.5 DD

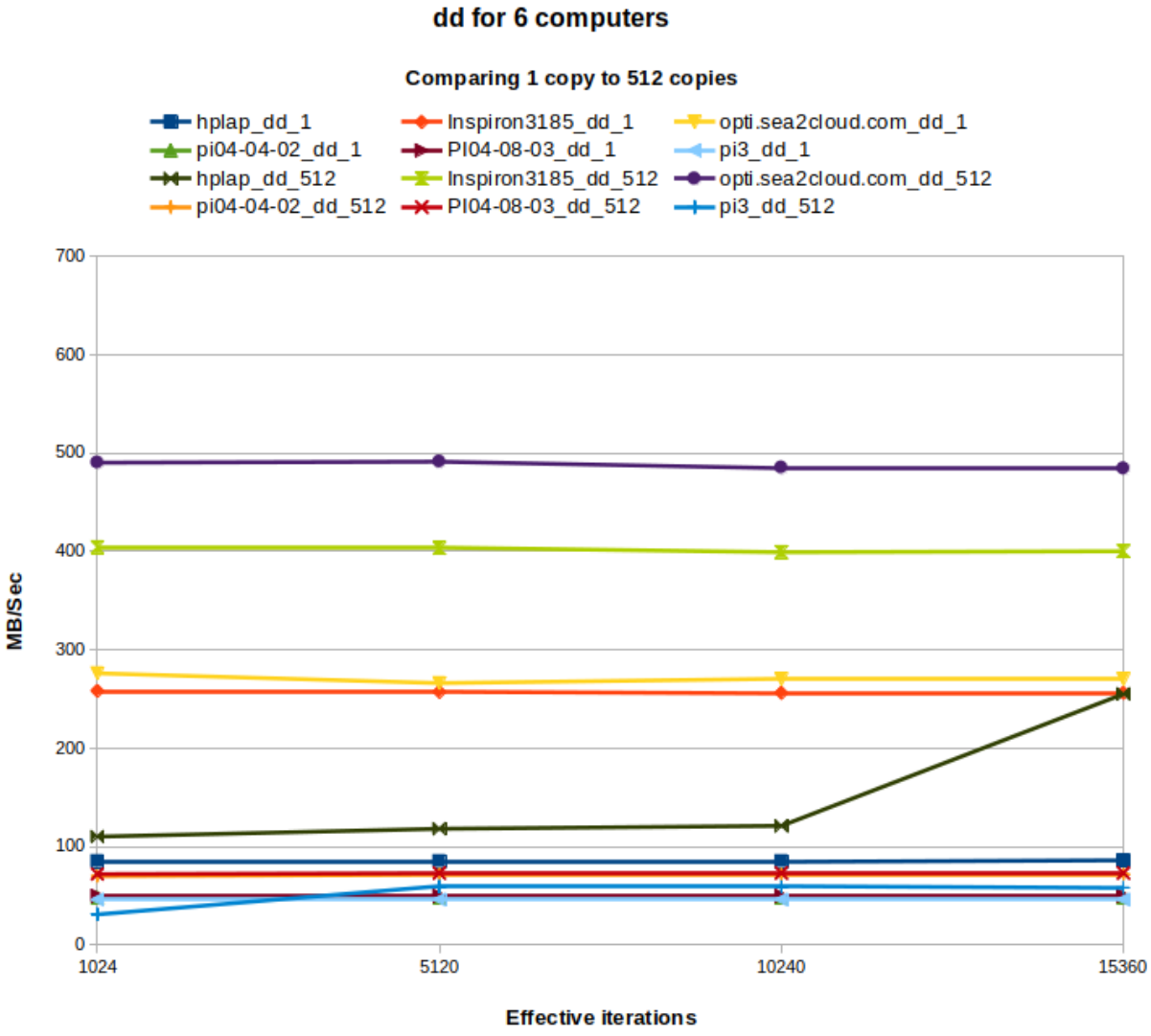

Figure 5: dd results

This Figure 5: dd results illustrates the processing rate of dd for reading a file and flushing it to "/dev/null." It is interesting to note that the throughput rate for dd is slightly slower than the throughput rate shown in Figure 1: b2Sum Results. I can only speculate that it is because the "dd" program copies the input data to the output buffers before realizing that the buffers will be flushed to "/dev/null." Perhaps there is an optimization here for the "dd" application to use it to measure "read" rates and eliminate the actual "copy" to the output buffer if the output device is "/dev/null."

I have no idea why the ancient HP laptop that has been collecting dust in my office suddenly was able to produce an astounding result at an "Effective Iteration" of 15Kib on 512 copies of the dictionary.

6.6 OPTIPLEX980 ALL TESTS

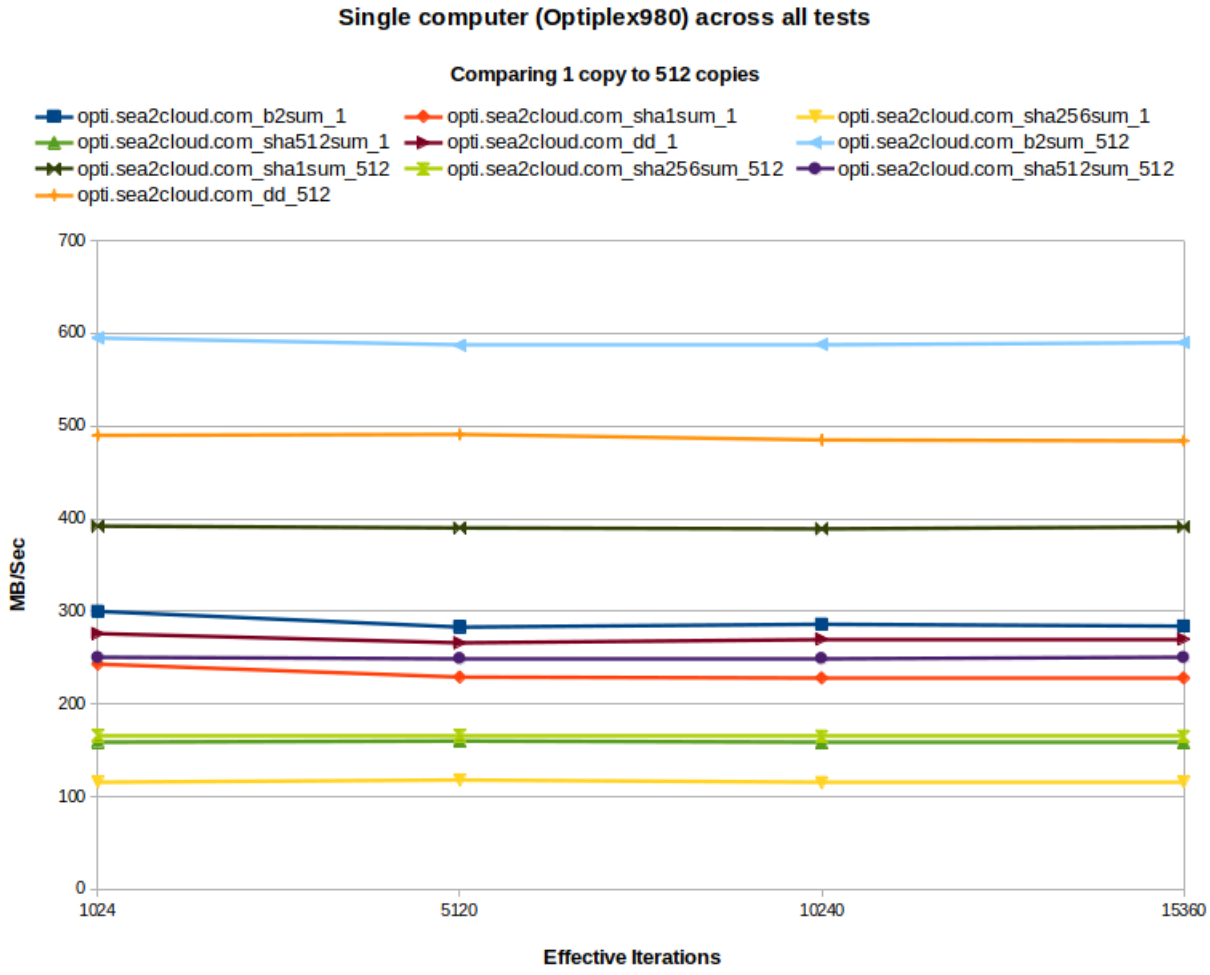

Figure 6: Optiplex980 results

The Optiplex980 machine, despite its age,[16] is actually the fastest machine that I currently own. Part of the speed is due to the fact that the primary drive is an M2 Flash device for the operating system and /tmp storage. The four core CPU also has a respectable clock speed.

[16]The machine is one that my brother recycled from a customer that was retiring the machine in favor of a faster machine. My acquisition cost was $75 for shipping. I then upgraded the primary drive to be and M2 Flash device rather than spinning disk.

6.7 PI04-08-03

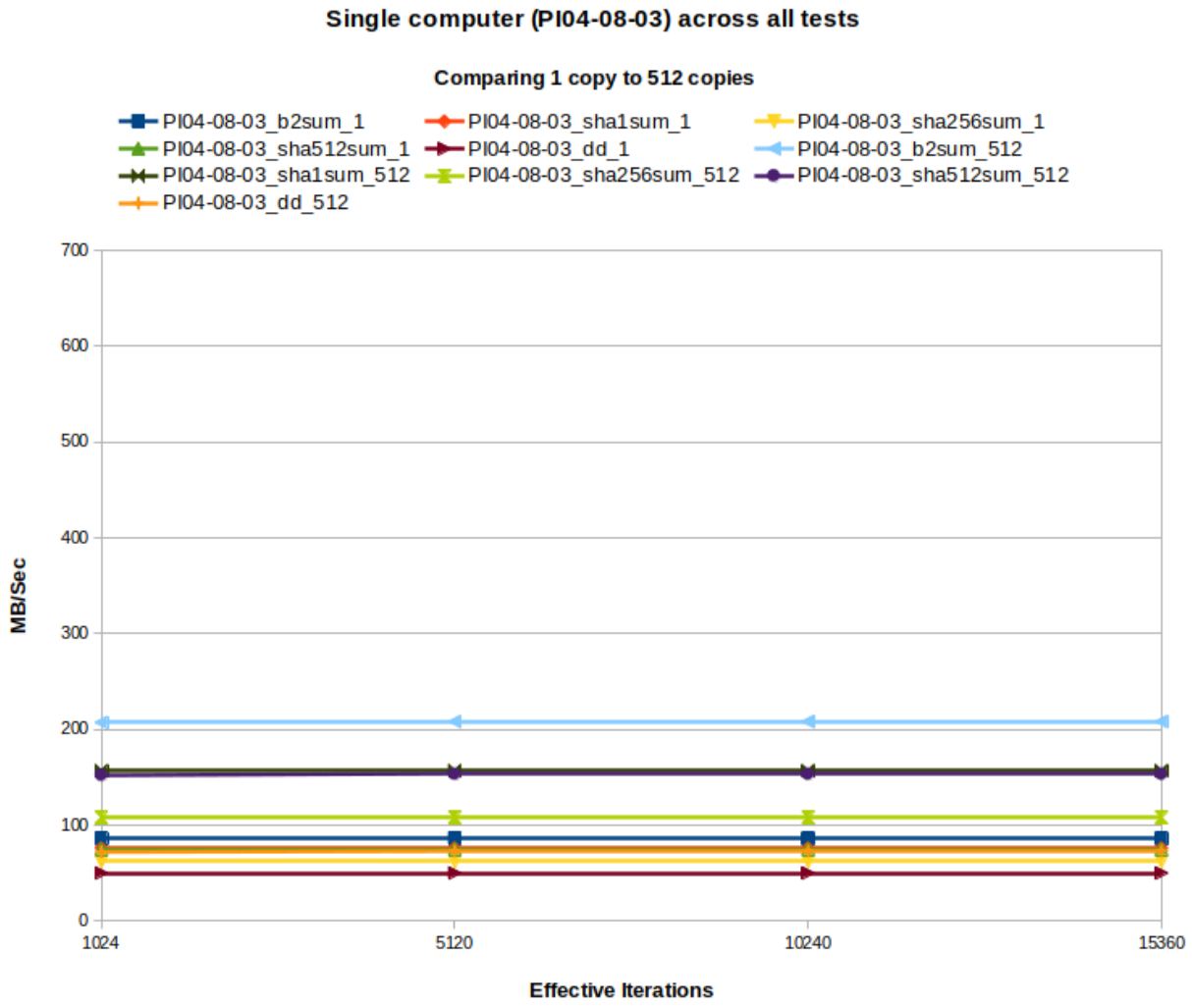

Figure 7: Raspberry PI 4 w/8GB results

Although these results look anemic compared to the ancient Optiplex980, the next graph shows that this machine is far better than its Bogomips rating of 108 (versus the Optiplex980 Bogomips rating of 6834.5). The machine is definitely not 1/60th of the performance of the beefier machine as the next two sections will show.

6.8 RELATIVE PERFORMANCE RATIOS

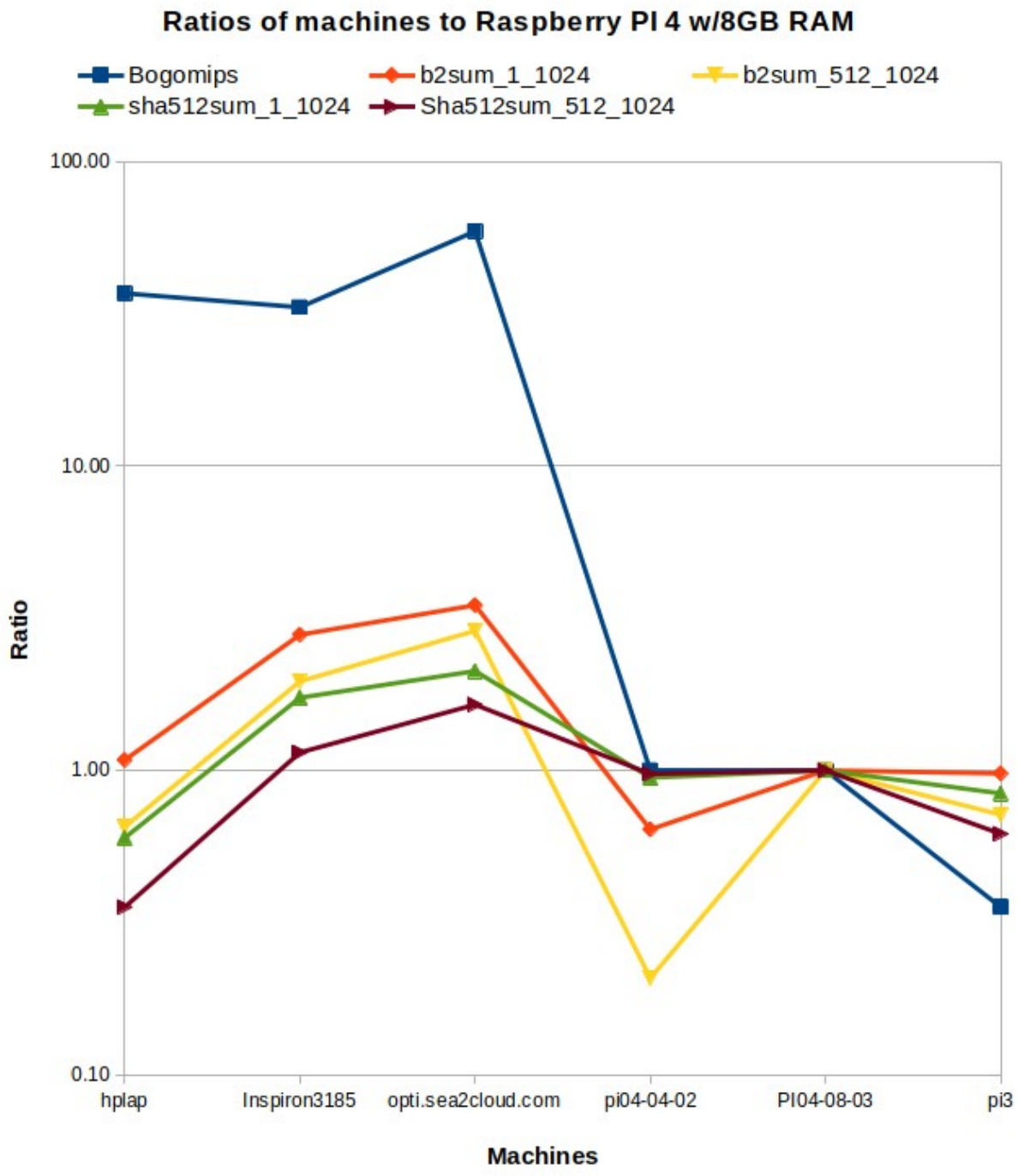

Figure 8: Ratios of machines

In this Figure 8: Ratios of machines I have shown the relative ratios of the various machines to the Raspberry PI 4 with 8GB of RAM. To avoid data overload I have shown only the Bogomips, b2sum (1 copy and 512 copies) and sha512sum (1 copy and 512 copies) results.

	Hostname					
Test	**hplap**	**Inspiron3185**	**opti.sea2cloud.com**	**pi04-04-02**	**PI04-08-03**	**pi3**
Bogomips	3992.77	3593.13	6384.5	108	108	38.4
b2sum_1_1024	93	240	300	55	86	84
b2sum_512_1024	135	406	595	43	207	148
sha512sum_1_1024	45	130	159	71	75	63
Sha512sum_512_102	54	174	250	148	152	94

Table 2: Selected performance results

This Table 2: Selected performance results shows the performance results that were used in calculating the performance ratios shown in Table 3: Ratios for selected performance results. These results were normalize to the results of the Raspberry PI 4 with 8GB of RAM.

	Hostname					
Test	**hplap**	**Inspiron3185**	**opti.sea2cloud.com**	**pi04-04-02**	**PI04-08-03**	**pi3**
Bogomips	36.97	33.27	59.12	1.00	1.00	0.36
b2sum_1_1024	1.08	2.79	3.49	0.64	1.00	0.98
b2sum_512_1024	0.65	1.96	2.87	0.21	1.00	0.71
sha512sum_1_1024	0.60	1.73	2.12	0.95	1.00	0.84
Sha512sum_512_102	0.36	1.14	1.64	0.97	1.00	0.62

Table 3: Ratios for selected performance results

6.9 MEANS OF TESTED RESULTS

This Figure 9: Means of ratios for non-Bogomips results illustrates the aggregate performance ratios of the various machines tested using the cryptographic hash and dd results generated in testing. I have computed both the aritmetic mean and the geometric mean for the ratios. These results are more consistent with my intuition about the relative performance of the machines than the Bogomips ratings.

	Arithmetic Mean	Geometric Mean
hplap	81.69%	70.76%
Inspiron3185	259.35%	223.10%
opti.sea2cloud.com	315.55%	280.50%
pi04-04-02	85.55%	79.32%
PI04-08-03	100.00%	100.00%
pi3	74.60%	72.64%

Table 4: Means of Ratios for non Bogomips performance results

7. Conclusion

Although the Raspberry PI 4 with 8GB of RAM appears to be 2-3 times slower than the fastest machines that I tested, it shows that the machine has quite respectable peformance. I was surprised to note that the peformance rivaled the laptop from HP that I had used in 2015. The system is definitely not as snappy as the ancient Optiplex980.

As always, just as in gas mileage estimates, yourF mileage may vary depending on your actual use. My primary use for all of my machines is a combination of software development (writing code using gvim), storing code at github, making binary programs and using office applications like Libreoffice. These are not the intensive CPU, graphics and I/O applications that modern games use. But for ordinary desktop applications and web browsing the Raspberry PI is more than adequate.

7.1 SURPRISE #1

As I noted the Raspberry PI appears to have the same class of performance as the 2015 HP laptop. However, it has significantly lower price (~$100) vs (~$600). It has higher resolution graphics (dual 4K HDMI output). Best of all the 2015 machine had an extreme heating problem. Even a simple application like a Skype video chat would cause the system to overheat and shutdown after a short period of time. That is why the machine was retired.

7.2 SURPRISE #2

Looking at the performance of the Raspbery PI 4 with only 4 GB of RAM, It shows that the RAM does make a difference. An identical system board with 1/2 fhe memory shows an approximately 20% drop in peformance. That said, I use this machine every single day. The 4GB machine is attached to my living room large screen TV (I had to reduce the resolution to read the text from across the room). This document was prepared with LibreOffice Writer and the accompanyin chart and tables with LibreOffice Calc on that machine. I am using NFS to access the disk drives on the Optiplex980 machine in my office.

The machine with 8GB RAM is connected to the TV in my bedroom. For my day to day operations, there is not a perceptible difference between the two machines. They are not as snappy as the deskside machine in my office, but perfectly usable. The biggest slowdowns are with apps that abuse Java for graphics rendering (e.g., Lastpass takes seconds to bring up the password screen).

7.3 SURPRISE #3

Note that the Raspberry PI 3 machine that I tested (it was one of the first Raspberry PIs that I purchased) only has 1 GB of RAM. It is NOT running Ubuntu. It is running the 64 bit Raspberry PI OS. As these tests show, it is running it quite well. For selected tests it even outperformed the 4 GB Raspberry PI 4, despite a slower CPU and smaller memory. Apparently the Raspberry PI OS has a smaller memory footprint than Ubuntu. I am not going to try to run

Ubuntu on this machine, but it ran a number tests very well. This seems to be an ideal machine for a heatless server.

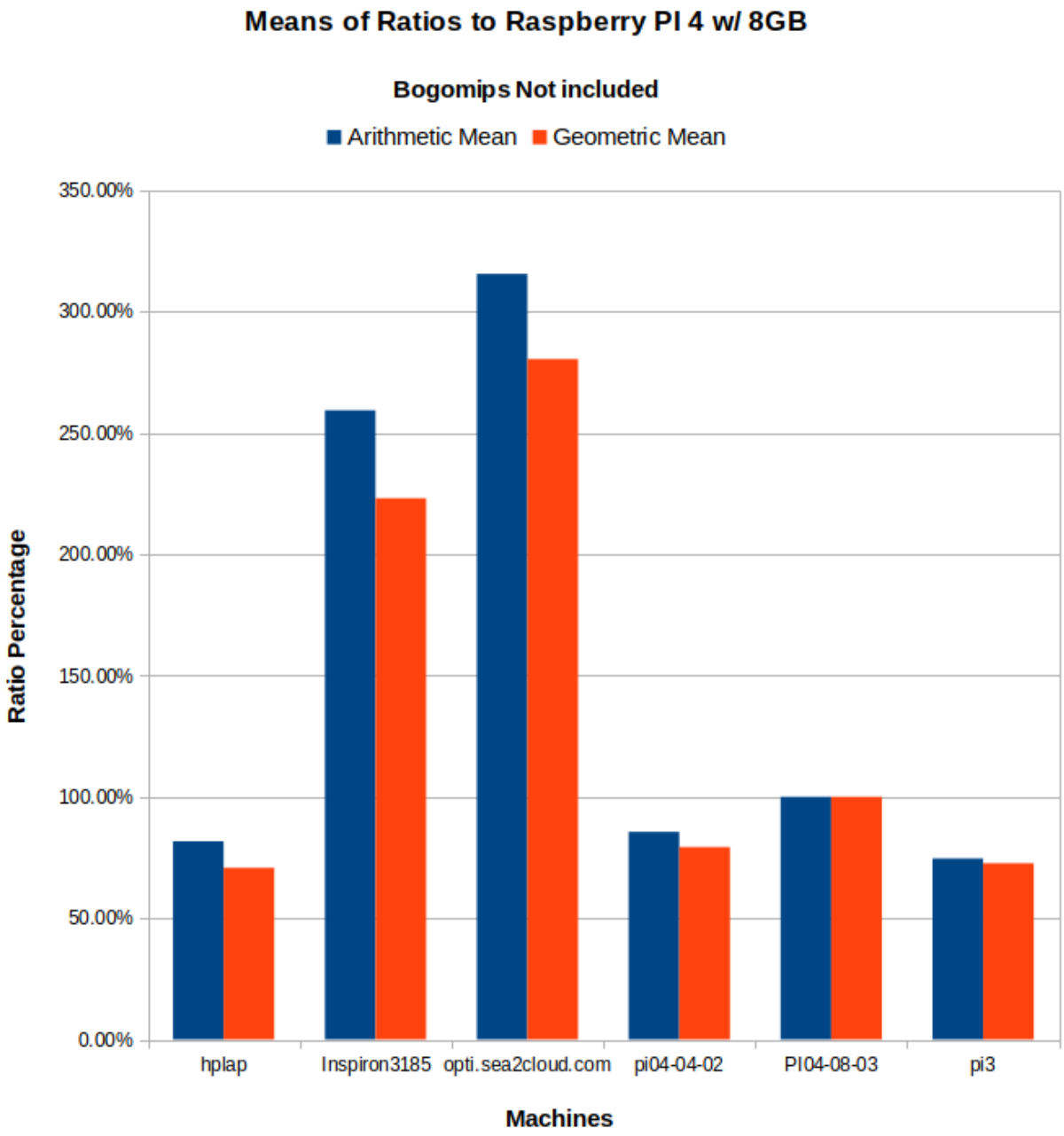

Figure 9: Means of ratios for non-Bogomips results

8. THOUGHTS FOR FURTHER INVESTIGATIONS

All of the source code for this testing (BASH scripts) can be found in a public repository on github:

https://github.com/sailnfool/sysperf

8.1 OTHER DICTIONARIES

I would be delighted to hear if someone gets different results by using a different dictionary.

8.2 RANDOM DATA

Another possibility is to use a random number generator to create the input file which is processed by the programs.

8.3 MOVE /TMP TO ZRAM

This is the one that I suspect would be the most promising. I have been reluctant to move /tmp into zram since my systems are largely memory constrained and I don't have a good algorithm to decide how to divide up space between SWAP and TMP. With the advent of SNAP in Ubuntu, I find the /tmp is littered with long lasting directories used by processes. Some investigation into the amount of /tmp space needed by Ubuntu (or other distributions) is probably warranted. Moving /tmp into zram could have large performance consequences.

9. COMMAND -H OUTPUT

9.1 MCSPEED -H

```
mcspeed [-h] [-c <#>] [-l <language> ] [-n]
                    [-s <hashprogram>] [-w <#>] <nicenumber>
              Provide a nicenumber to control the iterations of the number of
              times that we take the sha256sum of the local dictionary. This
              will report the architecture of the machine, the size of the
              dictionary, the number of iterations, bogomips, time per hashing
              10Mib characters as normalization
        <nicenumber>      A number or a number in the format "1M" ori
                    "1Mib" to represent:
                     1 M (Mbyte) 1000000
                    or 1 Mib (Mbibyte) 1048576
                    or 1 Pib (Pbibyte) 1125899906842624
                    or other bibyte prefix (e.g., eta or zeta)

                    For a complete table of nice suffixes and values
                    mcspeed -v -h
        -c      <#> the number of copies of the dictionary that are
                    concatenated together to diminish the open/close processing
                    for the opening and closing of the input file. The default
                    number of copies is 1. Suggested values are
                    between 128 and 512 copies. See also -w below
        -h          Print this message
        -l          <language> the name of an alternate dictionary
```

```
        -n              Send the hash output to /dev/null
        -s      <hashprogram>    Specify which cryptographic hash
                        program to use valid values:
                        b2sum; sha1sum; sha256sum; sha512sum
                        or the special case "dd" which helps measure
                        the file operations (open, read, write)
                        without the cryptographic processing
        -v              turn on verbose mode, currently ignored
        -w      <#>     the divisor used to provide a "wait" time for the
                        copies operation to complete. When making (e.g. 512)
                        copies tof dictionary that is ~1MB in size, it takes
                        time for that operation to complete so that it does
                        not delay the "timing" function. Using the
                        graphical gnome tools that display the
                        write operations for the system, I have experimented
                        with this on a Raspberry PI 4 and it appears that it
                        takes a divisor value of between 8 and 20. Your
                        mileage may vary so perform your own testing.
```

9.2 MWRAPPER -H

```
mwrapper [-h] [-c <#>] [-w <#>] [-d <suffix>] [-f <testname>i] [-n]
                [<maxiterations> [<increment>]]
                <maxiterations> (default 50)is the number
                of times that the test is performed for each hashtype. Note
                is multiplied by 1024
                <increment> (default 10) is the number of
                of iterations to bump the test count by each time
        -c      <#>     The number of copies of the input file to concatenate
                        together to minimize the open/close overhead.
                        Note: the number of copies is appended to the
                        suffix in order to distinguish both scripts
                        and results.
        -d      <suffix>        is the string just prior to the .csv extension
                        that can uniquely identify the generate script. The
                        default is normally the current date time
                        (e.g. 20220226_1916)
        -f      <testname> (default ldap.sea2cloud.com_20220226_1916) is
                        the name of the CSV file where the results are
                        tabulated.
        -h              Output this message
        -n              Send the hash output to /dev/null
        -s      <hashprogram>    Specify which cryptographic hash
                        program to use valid values:
                        b2sum; sha1sum; sha256sum; sha512sum
                        or the special case "dd" which helps measure
                        the file operations (open, read, write)
                        without the cryptographic processing
        -w      <#>     the divisor used to provide a "wait" time for the
                        copies operation to complete. When making (e.g. 512)
                        copies tof dictionary that is ~1MB in size, it takes
                        time for that operation to complete so that it does
                        not delay the "timing" function. Using the
                        graphical gnome tools that display the
                        write operations for the system, I have experimented
                        with this on a Raspberry PI 4 and it appears that it
                        takes a divisor value of between 8 and 20. Your
```

mileage may vary so perform your own testing.