Portable hotplugging: NetBSD's uvm_hotplug(9) API development

Santhosh N. Raju
santhosh.raju@gmail.com

Cherry G. Mathew
cherry@NetBSD.org

Abstract

NetBSD has quietly re-engineered its virtual memory subsystem over the years via `uvm(9)`. `uvm(9)` is very portable—to the point where usermode kernels, paravirtualised kernels and monolithic kernels all share the same machine independent (MI) VM code. (The notable exception being `rumpkernel(9)`, which brews its own stubs).

One sticky point in all these configurations is the early boot time system memory segment management. NetBSD, still "managed" this with a static array.

We show how we transformed this static implementation into a dynamically managed key/value pair system, almost entirely in userspace, using Test Driven Development methodologies on NetBSD's automated testing framework.

We further show how this enabled us to measure the performance of our changes, within the limitations of NetBSD's testing framework by integrating load testing into our userspace development methodology.

1 Introduction

The virtual memory subsystems of many modern server operating systems have support for hotplugging memory [6, 16]. This enables one to dynamically add or remove memory without the need to power off the system. While NetBSD's virtual memory manager `uvm(9)` [15] is quite robust and portable, it lacked this ability to hotplug memory segments on the fly until recently.

In this paper, we look at the strategies that we used to implement such a hotplug system in NetBSD via Test Driven Development [12] (TDD). We explore some of the challenges as well as implementation details that eventually led to making `uvm_hotplug(9)` [14] a feature in NetBSD-CURRENT as of writing this paper.

We also perform a side by side benchmark of the old API against the new API and show the results.

2 Background

In order to understand how `uvm_hotplug(9)` was implemented, we first need to understand how the current `uvm(9)` subsystem is limited in NetBSD.

The current `uvm(9)` implementation in NetBSD is unaware of changes to (increase or decrease) physical memory size. This is due to the way the memory pages are statically allocated during kernel boot time and a list of free pages is maintained in a static array along with a set of properties (tags).

The static array implementation of `vm_physseg` which gets initialised during boot time is defined as follows in `uvm_page.c`:

```
struct vm_physseg vm_physmem[VM_PHYSSEG_MAX];
int vm_nphysseg = 0;
#define vm_nphysmem     vm_nphysseg
```

This implementation prevents any additional run-time modification either by expanding or collapsing this array. As we proceed through this paper, we will show how this naive array implementation was replaced by a more robust R-B tree implementation via the `rbtree(3)` [11] API which would become the basis for implementing the `uvm_hotplug(9)` feature.

3 Sanitising for `uvm_hotplug(9)`

In this section we describe the strategies we used to sanitise the parts of the existing `uvm(9)` API relevant to "hotplug", so that we could apply TDD methodology to design and implement the `uvm_hotplug(9)` API.

3.1 The reference

TDD, as the name suggests, is a development methodology where test code is written before the actual implementation of new functionality. Applying this methodology to the NetBSD kernel was quite a challenging task.

Most of the code we aspired to work with within uvm(9) was historically written without unit testing in mind and no tests had been written to validate the existing implementation.

We first developed an idealised expected API meant to represent how the hotplug API should look. We then wrote the tests for this API, which now acts as the set of baseline tests which would act as the starting point of the uvm_hotplug(9) development.

Both these tests and the idealised API were buildable via ATF [1] but was not written so that the build.sh [7] would build them.

3.2 The separation

Isolating the existing API that deals with segment handling in uvm(9) was essential for the rest of the TDD methodology to work. We spent quite some time going through uvm_page.c trying to figure out sections of code that were significant to our work.

After this we separated the relevant functions, structures, variables, etc. into a header and source file. Now segregated into uvm_physseg.c and uvm_physseg.h it was time to deal with the relevant machine dependent (MD) parts of the code.

3.3 Exposing the API

Our initial changes to MD mostly focused on **amd64** and **i386** since these where the architectures that were readily available to us. During this process we identified various sections of code that were then converted to "utility" functions and exposed via the uvm_physseg.h file.

This was necessary for us to achieve testing code in isolation, which is quite important when it comes to unit testing.

To keep a disciplined API, we followed these steps:

- Kept structures that needn't be exposed globally to the users in a .c file. This would force future consumers of the API to not make assumptions about implementation.

- The .h file nicely exposes all the "valid" operations that can be done on the various opaque structures that is used in this API. In terms of object oriented concepts these would be "getters" and "setters" along with the functionality the API can provide.

This refactoring effort resulted in actual buildable and bootable code for both the architectures that we were working with.

3.4 Testing in userspace

Once we had proper, buildable code, the next job was to get the reference tests we had written for our reference API to work with the actual existing API.

In order to get the kernel code to work in userspace we did the following:

- Included the .c file as part of the ATF test.

  This made the entire API exposed to userland ATF, and enabled isolated testing of the various functions exposed in the .h file.

- Stub / re-implemented kernel API calls.

  This was important, since many of the included in uvm_physseg.c for example used functions that were only accessible when running within the kernel, for example kmem_alloc() [5] or kmem_free() and these could not be accessed from witin userland. One way we implemented this was to have a separate include that would import these functions into the MD header file so that they did not get included in the ATF tests causing build errors. We then re-implemented these functions with User land wrappers of equivalent semantics. For example malloc() or free(). Of course this was only a workaround, but it did provide sufficient isolation for unit / functional testing.

- Stub / re-implemented dependent API calls.

  Similar to above, the file we were including may be dependent on other APIs that the system uses for example uvm_physseg.c depends on certain calls from uvm_page.c A question we always had to remind ourselves about was, whether to stub them or re-implement the whole function ? Our take on this was to stub them with placeholder return values for dependant tests. A minimalstic re-implementation was demanded for sometimes, just enough so that the tests could make progress.

After this we had a set of working tests that could test the correctness of the current static array implementation and this would act as our starting point for implementing the R-B tree based memory segment handling.

4 Design and implementation

Here we have a look at the design considerations for a dynamic data structure so that we have hotplugging as a feature in NetBSD current and the implementation details of this.

4.1 From static to dynamic

For a data structure where dynamic insertion and deletion of segments could be made, we chose to use the R-B tree, based on the `rbtree(3)`, which comes as a part of the standard C Library.

Our choice of data structure was influenced by the following:

- We no longer have to worry about maintaining a sorted order, in-order traversal of the R-B tree always results in sorted order retrieval. What makes this easier is the macro provided by the API `RB_TREE_FOREACH()` which goes through each of the node in sorted order.
- No more multiple strategies for maintaining the segments. The current one provides 3 separate strategies *RANDOM*, *BSEARCH* and *CONTIG* depending on how you want to keep the segments. With the choice of R-B tree implementation, all of the strategies would boil down to *BSEARCH*.
- Less code clutter, since we are now dependent on the R-B tree API to do the inserts / removals / retrievals, we do not need to worry about writing code to maintaining and managing the data structure itself, hence making a reliable and robust code base which is less prone to errors.
- Neat and clean API, compared to `queue(3)` [9] and `tree(3)` [13], the `rbtree(3)` API is cleaner and neater to read and to implement.

4.2 Design challenges

The transition was not easy, we did face some issues while porting. Some changes were made to the existing system to make the porting easier.

- The current handle for accessing a segment is the index of the array `vm_physmem[]` which is of type `int`. For R-B tree the proposed equivalent handle would be `struct vm_physseg *`, this also means "`for()` loops" that were created for iterating the array would be needed to re-written carefully so as not to break the system.
- In order to ease the transition, we introduced a new abstraction for the above mentioned handle called `uvm_physseg_t` which will be a `typedef` over the existing handle and then introduce enough utility functions that can help replace the current assumed properties of the int type which are used in various sections through UVM.
- Since we are modifying a fundamental part of the operating system, this implies every single architecture port of NetBSD [8] (which is 78 at the time of writing this) would have to be touched.
- A question at the back of our minds was if, finally, with the new code, what the performance implications would be.

4.3 Implementing the R-B tree

Here we have a quick overview of how the R-B tree based implementation has affected the API.

- Introduction of `uvm_physseg_t`

  The new abstraction `uvm_physseg_t` now encapsulates the handle that is used to access the current "segment" or "node", this concept is also backported into the existing static array implementation, so that the consumers of the `uvm_physseg` API do not break.

- Introduction of iterators

  With the introduction of a new handle, we cannot assume we can loop through an array using just integers like we did in the current code base.

  The static array implementation iterating through the segment looks like this:

```
for (lcv = 0 ; lcv < vm_nphysmem ; lcv++) {
  seg = VM_PHYSMEM_PTR(lcv);
  freepages += (seg->end - seg->start);
}
```

  With the R-B tree changes, the above loop looks like this:

```
for (bank = uvm_physseg_get_first();
     uvm_physseg_valid(bank);
     bank = uvm_physseg_get_next(bank)) {
     freepages += uvm_physseg_get_end(bank) -
                  uvm_physseg_get_start(bank);
}
```

  They are more agnostic and do not assume an underlying data structure for the "segment". This means that all of these changes need to propagate through the MI and MD Part of the code

- Introduction of `uvm_physseg_valid()`

  This is a function call that can be used to check the validity of a given segment. And returns a boolean depending on the validity of the passed segment.

5 Testing uvm_physseg via ATF

We had the baseline set of ATF tests written for the original static array implementation, since the rbtree(3) implementation should behave exactly the same as the static array, we just had to make sure that the tests which were passing for static should also be passing for rbtree(3). This provided a rather easy way to test the correctness of our API even before we started to build the code base to do an actual test. Overall this did reduce considerably the amount of time we needed to spend to make sure the old and the new implementation were working as expected without the need to worry about the details.

Despite this there were some interesting edge cases that came along

5.1 Prototype of uvm_page_physload()

One such interesting case was the behaviour of uvm_page_physload(), this function was originally designed to plug in segments of memory range during boot time. Hence if any errors happened it would generally print a message and make the kernel panic. Since this was the way it was expected to work, it was fine for it to return nothing (void) after its execution. The original prototype looked like this

```
void
uvm_page_physload(paddr_t, paddr_t, paddr_t,
                  paddr_t, int);
```

However this was a bit of a hurdle when it came to writing unit tests. One of the biggest annoyances was getting a handle to the segment that was plugged in. Since uvm_page_physload() would not return anything, the only way to get a reference to the first inserted segment would be to do a uvm_physseg_get_first(). Most of the initial unit tests written assume that we fetch the desired handle using uvm_physseg_get_first() function, but this was only a work around, since this is not a reliable approach in this methodology.

```
uvm_physseg_t
uvm_page_physload(paddr_t, paddr_t, paddr_t,
                  paddr_t, int);
```

The change helped in removing unwanted assumptions in our ATF tests, making for cleaner and compact test cases.

5.2 Immutable handles

The introduction of a handle uvm_physseg_t which is used to access the data structure that keeps physical segment information, we came across a failing ATF test case in the static array implementation which was working as expected in the R-B tree implementation. For the static array implementation we were using the VM_PSTRAT_BSEARCH strategy.

The specific test case in question is uvm_physseg_get_prev which had the segments being inserted into the system out-of-order, this meant that the page frames of the segments that were inserted in chunks were not in a sorted order. So the first chunk may be in the *256* to *512* range and the second chunk was in the *0* to *256* range. The same chunks if inserted in sorted order resulted in a passing test.

After some amount of debugging we found out why this rather seemingly trivial change in the order of insertion was causing the tests to fail, this was a consequence of changing the way the segment was being referenced in the underlying data structure from static array implementation to R-B Tree implementation.

The **handle** in the static array is the **index** of the array in which the segments are stored. So when segments are inserted out of order, like in the example above the content of the index is likely to change causing this issue. In case of the R-B tree, the handle is a **pointer to the segment** itself, which does not change when a new segment is inserted or deleted.

In order to separately identify this property of mutability we added a new test case in ATF uvm_physseg_handle_immutable which should always fail for static array implementation.

This is important to notify the users of the old API and new API about the potential pitfall of assuming the integrity of the handle when writing new code.

6 Booting the kernel

Once we had finished the first successful build of NetBSD with uvm_hotplug(9) enabled, it was time to boot it and see if the changes actually work.

6.1 The init dance

Despite a successful compile of the hotplug enabled kernel, the first boot led to a panic. We quickly identified that kmem(9) is not available until uvm_page_init() has done with all the initialization.

We went for maintaining a minimal "static array" whose size is VM_PHYSSEG_MAX and once the init process is over, switch over to the kmem(9) allocator.

In order to achieve this couple of wrapper functions that exist only within the realm of uvm_physseg.c called uvm_physseg_alloc() and uvm_physseg_free() were introduced.

We managed to write up the test cases for these first, hence allowing for a smooth implementation of the

switching between static array and kmem(9) based allocator to store the R-B Tree nodes.

6.2 Fragmentation of segments

One of the more interesting challenges we came across when developing the API was the handling of the "pages" array within the segment structure. The pages array, pgs[] holds the pages for a given segment. Unplugging a segment is bound to fragment this pages array which is currently allocated by kmem(9) allocator.

In order to overcome this issue we came up with an idea to use the extent(9) memory manager to manage the pages array.

The implementation of extent(9) [3] into uvm_hotplug(9) was also done with extensive ATF tests that helped us out in minimising the downtime from debugging the code when we finally managed to boot the kernel with these features enabled.

7 Performance evaluation

Once we had a stable kernel up and running, we needed to benchmark how the API performed. Since we had no real world implementation of uvm_hotplug(9) yet, we decided to leverage the ATF framework to get this job done for us.

7.1 Designing the test framework

The most frequent operation that is impacted by our change is the look up call uvm_physseg_find() which now searches through a R-B tree instead of a static array. In order to simulate this we copied over the PHYS_TO_VM_PAGE() macro and the related code from uvm_page.c and then we wrote some ATF tests that would load some memory segments, followed by multiple calls to PHYS_TO_VM_PAGE() that would search for random addresses within the plugged in segments.

```
for(int i = 0; i < 100; i++) {
        pa = (paddr_t) random() %
             (paddr_t) ctob(VALID_END_PFN_1);
        PHYS_TO_VM_PAGE(pa);
}
```

The above snippet does tests for 100 calls like mentioned in the comment for it. This methodology is not a perfect load test since there is a call to random() which will cumulatively add up to the runtime of the function we are trying to load test.

After some tweaking around we managed to write up the tests varying from 100 calls to 100 Million calls and then evaluate the time for them. For these tests we added a ATF_CHECK_EQ(true, true) at the bottom of the test indicating the test will never fail.[1]

A second type of performance test we came up with is a search done on highly fragmented memory segments that have been unplugged from a fixed chunk. This requires the boot process to be faked since we need to invoke the uvm_physseg_unplug() to fragment the memory. After this 10 Million calls are made to the PHYS_TO_VM_PAGE() macro and the memory segment sizes were varied from 1 MB to 256 MB on a VirtualBox instance of NetBSD that had a total of 512 MB to spare. This test is specific to R-B Tree implementation and cannot be run for static array implementation, since VM_PHYSSEG_MAX will limit the amount of fragments that can happen to the array, and since this is a very small value like 32, it would not make much sense to test it out.

An example run of these tests with the standard atf-run piped through atf-report will have a similar output.

```
t_uvm_physseg_load (1/1): 11 test cases
    uvm_physseg_100: [0.003286s] Passed.
    uvm_physseg_100K: [0.010982s] Passed.
    uvm_physseg_100M: [8.842482s] Passed.
    uvm_physseg_10K: [0.004398s] Passed.
    uvm_physseg_10M: [0.954270s] Passed.
    uvm_physseg_128MB: [2.176629s] Passed.
    uvm_physseg_1K: [0.002702s] Passed.
    uvm_physseg_1M: [0.094821s] Passed.
    uvm_physseg_1MB: [0.984185s] Passed.
    uvm_physseg_256MB: [2.485398s] Passed.
    uvm_physseg_64MB: [0.914363s] Passed.
[16.478686s]

Summary for 1 test programs:
    11 passed test cases.
    0 failed test cases.
    0 expected failed test cases.
    0 skipped test cases.
```

8 Benchmark Results

We ran the tests in an *i386* instance of VirtualBox running NetBSD.

The values used in the results section are an average of the 100 runs, we also show the Minimum and Maximum values. In addition to this we also, ran a dummy run on the random() function call without the PHYS_TO_VM_PAGE() translation happening. This was done to find out how much additional time was taken up by the random() [10]. This number did become significant for very large values of the looping test as we will see in the results section.

8.1 Calls to `PHYS_TO_VM_PAGE()`

Overview of results for different loop counters to the `PHYS_TO_VM_PAGE()` call. [2] [3]

This test was done for both the static array implementation as well as the R-B Tree implementation.

Test Name	Average	Minimum	Maximum
uvm_physseg_100	0.004599	0.003286	0.010213
uvm_physseg_1K	0.002740	0.001991	0.005747
uvm_physseg_10K	0.003491	0.002836	0.007941
uvm_physseg_100K	0.011424	0.009388	0.017161
uvm_physseg_1M	0.093359	0.079128	0.138379
uvm_physseg_10M	0.892827	0.813503	1.172205
uvm_physseg_100M	8.932540	8.434525	11.616543

Table 1: Comparison of average, minimum and maximum for R-B tree implementation.

Test Name	Average	Minimum	Maximum
uvm_physseg_100	0.004714	0.003511	0.013895
uvm_physseg_1K	0.002754	0.002088	0.005318
uvm_physseg_10K	0.003585	0.002666	0.005271
uvm_physseg_100K	0.011007	0.009199	0.016627
uvm_physseg_1M	0.086208	0.076989	0.116637
uvm_physseg_10M	0.843048	0.782676	0.980598
uvm_physseg_100M	8.434760	8.128623	9.132065

Table 2: Comparison of average, minimum and maximum static array implementation.

For a more in-depth analysis, we plotted graph which gives a more magnified view of how the R-B tree implementation compares against the static array implementation over 100M calls to `PHYS_TO_VM_PAGE()`.

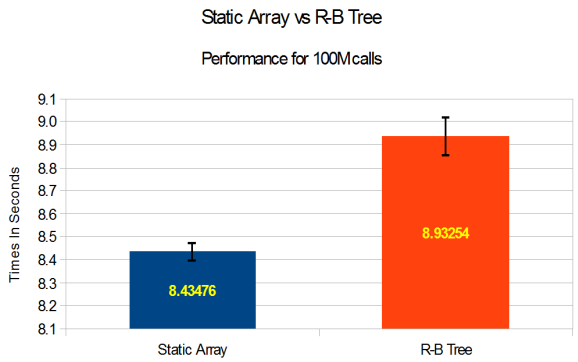

Figure 1: Static array vs R-B tree for 100M calls to `PHYS_TO_VM_PAGE()`

Clearly there is a 5.59% degradation in performance with the R-B tree implementation.

The tabular representation of the various statistics for the 100M Calls to `PHYS_TO_VM_PAGE()`. We calculated the Average, Standard Deviation (Population) and Margin of Error with a 95% confidence interval.

In a total of 100 runs, the `random()` function contributed to roughly 2.03 seconds for the average runtime, for a 100 Million calls to `PHYS_TO_VM_PAGE()`.

	Static Array	R-B Tree
Average	8.43476	8.93254
Standard Deviation	0.19331	0.41553
Margin of Error	±0.03789	±0.08144

Table 3: Comparison of the average, standard deviation and margin of error for the 100M calls to `PHYS_TO_VM_PAGE()`

8.2 Calls to `PHYS_TO_VM_PAGE()` after fragmentation

Number after test name indicates the amount of memory on which fragmentation was done by uvm_physseg_unplug(), memory was unplugged every 8 Frames starting from PFN 8.

After unplug was completed `PHYS_TO_VM_PAGE()` was called 10M (million) times for every test.

Test Name	Average	Minimum	Maximum
uvm_physseg_1MB	1.015810	0.941942	1.361913
uvm_physseg_64MB	0.958675	0.877151	1.279663
uvm_physseg_128MB	2.155270	2.024838	2.866540
uvm_physseg_256MB	2.550920	2.360252	3.736369

Table 4: Comparison of average, minimum and maximum execution times of various load tests with `uvm_hotplug(9)` enabled on fragmented memory segments.

9 Conclusions and future work

In this paper, we have presented a new API to allow hotplugging of RAM managed by NetBSD's uvm subsystem. Despite the fairly technical and intrusive details of the implementation itself, the key findings we would like to present are about method and motivation - Systems Programming can be made much less stressful by using existing Software Engineering techniques.

We also note the availability of general purpose APIs such as rbtree(3) and extent(9) in the NetBSD kernel, which makes implementation much less headache. For example, the presence of these APIs made the unplug implementation unremarkable.

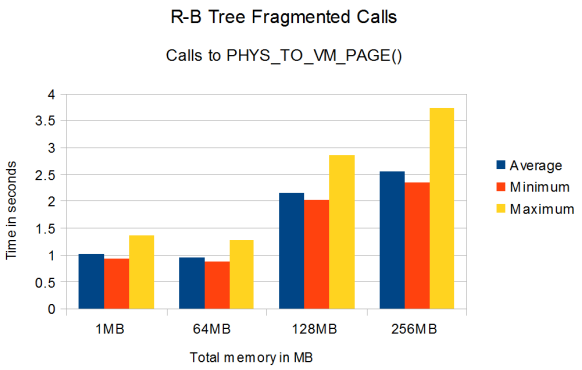

Figure 2: R-B tree performance for 10M Calls to `PHYS_TO_VM_PAGE()` after fragmentation at every 8 PFN

Note however that the API does not take responsibility for ensuring that pages that are unplugged are actually not in use. This is expected to be the responsibility of the respective driver writer [4]. This is a deliberate division of labour, which allows for clean abstraction. See the Xen balloon(4) [2] driver for an example of the API in action.

Finally, as Future work, we would like to encourage other NetBSD developers to use this API to write hotplug/unplug drivers for their favourite platforms with suitable hardware. We also encourage other BSDs to pick up our work - since this will clean up the current legacy implementations which are pretty much identical.

10 Acknowledgments

We would like to thank everyone who helped review drafts of this paper. Special thanks to Chuck Silvers for API review loops, and Nick Hudson for powering us through the code integration into NetBSD -current

References

[1] atf(7) - Automated Testing Framework, August 2010. See `http://netbsd.gw.com/cgi-bin/man-cgi?atf+7+NetBSD-current` for further information.

[2] balloon(4) - Xen memory balloon driver, July 2011. See `http://netbsd.gw.com/cgi-bin/man-cgi?balloon+4+NetBSD-current` for further information.

[3] extent(9) - general purpose extent manager, July 2014. See `http://netbsd.gw.com/cgi-bin/man-cgi?extent+9+NetBSD-current` for further information.

[4] uvm_hotplug(9) port-masters' FAQ., December 2016. See `https://wiki.netbsd.org/features/uvm_hotplug/` for further information.

[5] kmem(9) - kernel wired memory allocator, February 2016. See `http://netbsd.gw.com/cgi-bin/man-cgi?kmem+9+NetBSD-current` for further information.

[6] Linux Memory Hotplug, October 2007. See `https://www.kernel.org/doc/Documentation/memory-hotplug.txt` for further information.

[7] MEWBURN, L., AND GREEN, M. build.sh: Cross-building NetBSD, September 2003. See `http://www.mewburn.net/luke/papers/build.sh.pdf` for further information.

[8] Platforms Supported by NetBSD, December 2016. See `https://www.netbsd.org/ports/` for further information.

[9] queue(3) - implementations of singly-linked lists, lists, simple queues, tail queues, and singly-linked tail queues, October 2016. See `http://netbsd.gw.com/cgi-bin/man-cgi?queue+3+NetBSD-current` for further information.

[10] random(3) - better random number generator, June 2014. See `http://netbsd.gw.com/cgi-bin/man-cgi?random+3+NetBSD-current` for further information.

[11] rbtree(3) - red-black tree, August 2016. See `http://netbsd.gw.com/cgi-bin/man-cgi?rbtree+3+NetBSD-current` for further information.

[12] Test Driven Development, January 2017. See `https://en.wikipedia.org/wiki/Test-driven_development/` for further information.

[13] tree(3) - implementations of splay and red-black trees, July 2011. See `http://netbsd.gw.com/cgi-bin/man-cgi?tree+3+NetBSD-current` for further information.

[14] uvm_hotplug(9) - Memory hotplug manager, November 2016. See `http://netbsd.gw.com/cgi-bin/man-cgi?uvm+hotplug+9+NetBSD-current` for further information.

[15] uvm(9) - virtual memory system external interface, March 2015. See `http://netbsd.gw.com/cgi-bin/man-cgi?uvm+9+NetBSD-current` for further information.

[16] Hot-add memory support in Windows Server, October 2010. `https://msdn.microsoft.com/en-us/library/windows/hardware/dn613938(v=vs.85).aspx`.

Notes

[1] This is done because the test is NOT a check of correctness of the function being called, we assume the function works as expected when this test is running.

[2] The letter beside the number indicates the amount in thousands (K) or millions (M).

[3] All run times are recorded seconds.