Portable Hotplugging

A Peek into NetBSD's uvm_hotplug(9) API Development

Santhosh N. Raju
santhosh.raju@gmail.com

Cherry G. Mathew
cherry@NetBSD.org

March 11, 2017

Setting Expectations

What “Will” and “Will not” Be Covered?

What "Will" and "Will not" Be Covered?

What will **NOT** be covered...

- Usage of `uvm_hotplug(9)`
- Application of `uvm_hotplug(9)`
- Refer man page of `uvm_hotplug(9)` for that

What “Will” and “Will not” Be Covered?

What will **NOT** be covered...

- Usage of `uvm_hotplug(9)`
- Application of `uvm_hotplug(9)`
- Refer man page of `uvm_hotplug(9)` for that

So what I am going to talk about...

- Using TDD and how it was applied to `uvm_hotplug(9)` API
- Design changes in `uvm_hotplug(9)` and how they were implemented
- Some interesting edge cases in `uvm_hotplug(9)` development
- How we used `atf(7)` to do performance testing

Background

The Old Implementation

- Uses a static array (`vm_physmem[]`) to hold segments
- Maximum size of this array is defined in the macro `VM_PHYSSEG_MAX`

The Old Implementation

- Uses a static array (`vm_physmem[]`) to hold segments
- Maximum size of this array is defined in the macro `VM_PHYSSEG_MAX`
- Implementation can be seen in `uvm_page.c`

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

We trace our steps into showing you how we converted this **array** implementation to a `rbtree(3)` based implementation.

Sanitising for `uvm_hotplug(9)`

...Without loosing sanity

It took more than one step...

- Creating a **reference** API

...Without loosing sanity

It took more than one step...

- Creating a **reference** API
- **Separating** out the existing API

...Without loosing sanity

It took more than one step...

- Creating a **reference** API
- **Separating** out the existing API
- **Exposing** the now separated API

...Without loosing sanity

It took more than one step...

- Creating a **reference** API
- **Separating** out the existing API
- **Exposing** the now separated API
- **Testing** the API in userspace

Creating the Reference API

- There were no **Tests** to use as a reference
- We created an **Idealised** API to represent how the hotplug API should look.
- **Idealised** API now acted as the baseline for the ATF tests that should have been present in `uvm(9)`
- Chuck Silvers gave valuable feedback when we were making this **Idealised** API

Creating the Reference API

- There were no **Tests** to use as a reference
- We created an **Idealised** API to represent how the hotplug API should look.
- **Idealised** API now acted as the baseline for the ATF tests that should have been present in `uvm(9)`
- Chuck Silvers gave valuable feedback when we were making this **Idealised** API
  - **NOTE**: The “Idealised” API was not a part of the NetBSD build system. However the tests were buildable with `atf(7)`

Separating the Existing API

- Going through code mostly in `uvm_page.c` and some MD parts.
- Separated stuff into `uvm_physseg.c` and `uvm_physseg.h`
- Retrofitted relevant parts into various sections of **Idealised** API

Exposing the new API

- Kept structures that need not be exposed globally to the users in a `uvm_physseg.c` file
- The `uvm_physseg.h` file nicely exposes all the “valid” operations that can be done on the various opaque structures that is used in this API
- Exposed these utility functions via header file

Exposing the new API

- Kept structures that need not be exposed globally to the users in a `uvm_physseg.c` file
- The `uvm_physseg.h` file nicely exposes all the “valid” operations that can be done on the various opaque structures that is used in this API
- Exposed these utility functions via header file
- This refactoring effort resulted in actual buildable and bootable code

Testing in Userspace

Getting the kernel code to work in userspace

Testing in Userspace

Getting the kernel code to work in userspace

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

Testing in Userspace

Getting the kernel code to work in userspace

- Included the `uvm_physseg.c` file as part of the ATF test
- Stubbed / Re-implemented kernel API calls
- Stubbed / Re-implemented dependent API calls

Testing in Userspace

Getting the kernel code to work in userspace

- Included the `uvm_physseg.c` file as part of the ATF test
- Stubbed / Re-implemented kernel API calls
- Stubbed / Re-implemented dependent API calls
- This is similar to **Mocking** APIs

Testing in Userspace

Getting the kernel code to work in userspace

- Included the `uvm_physseg.c` file as part of the ATF test
- Stubbed / Re-implemented kernel API calls
- Stubbed / Re-implemented dependent API calls
- This is similar to **Mocking** APIs

An example of `kmem_alloc()` being stubbed

```
void *
kmem_alloc(size_t size, km_flag_t flags)
{
        return malloc(size);
}
```

Design and Implementation

From Static to Dynamic

We went for R-B Tree as the data structure for dynamic operations of insertion and deletion of memory segments.

From Static to Dynamic

We went for R-B Tree as the data structure for dynamic operations of insertion and deletion of memory segments.

- Implemented using the `rbtree(3)` part of NetBSD C Library.

From Static to Dynamic

We went for R-B Tree as the data structure for dynamic operations of insertion and deletion of memory segments.

- Implemented using the `rbtree(3)` part of NetBSD C Library.
- No worries about maintaining a sorted order. Made easier by `RB_TREE_FOREACH()`

From Static to Dynamic

We went for R-B Tree as the data structure for dynamic operations of insertion and deletion of memory segments.

- Implemented using the `rbtree(3)` part of NetBSD C Library.
- No worries about maintaining a sorted order. Made easier by `RB_TREE_FOREACH()`
- No more multiple strategies for maintaining the segments

From Static to Dynamic

We went for R-B Tree as the data structure for dynamic operations of insertion and deletion of memory segments.

- Implemented using the `rbtree(3)` part of NetBSD C Library.
- No worries about maintaining a sorted order. Made easier by `RB_TREE_FOREACH()`
- No more multiple strategies for maintaining the segments
- Less code clutter

From Static to Dynamic

We went for R-B Tree as the data structure for dynamic operations of insertion and deletion of memory segments.

- Implemented using the `rbtree(3)` part of NetBSD C Library.
- No worries about maintaining a sorted order. Made easier by `RB_TREE_FOREACH()`
- No more multiple strategies for maintaining the segments
- Less code clutter
- Neater and cleaner API, compared to `queue(3)` and `tree(3)`

Design Challenges

- **Handle** for accessing segment changed between static array and R-B Tree.

Design Challenges

- **Handle** for accessing segment changed between static array and R-B Tree.
- Index of array `vm_physmem[]` vs Pointer to `struct vm_physseg`

Design Challenges

- **Handle** for accessing segment changed between static array and R-B Tree.
- Index of array `vm_physmem[]` vs Pointer to `struct vm_physseg`
- Modifying a fundamental part of the operating system implies every single architecture port of NetBSD is affected. (77 at the time of writing this)

Design Challenges

- **Handle** for accessing segment changed between static array and R-B Tree.
- Index of array `vm_physmem[]` vs Pointer to `struct vm_physseg`
- Modifying a fundamental part of the operating system implies every single architecture port of NetBSD is affected. (77 at the time of writing this)
- What are the performance implications?

Implementing the R-B tree

- A new abstraction for the memory segment handles `uvm_physseg_t` was introduced

Implementing the R-B tree

- A new abstraction for the memory segment handles `uvm_physseg_t` was introduced
- Utility functions, to ease the transition
    - Before

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

Implementing the R-B tree

- A new abstraction for the memory segment handles `uvm_physseg_t` was introduced
- Utility functions, to ease the transition
  - Before

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

  - After

```
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);
}
```

Implementing the R-B tree

- A new abstraction for the memory segment handles `uvm_physseg_t` was introduced
- Utility functions, to ease the transition
  - Before

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

  - After

```
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);
}
```

- An interesting utility function to note is `uvm_physseg_valid()`

Testing `uvm_physseg` via ATF

Generic ATF Runs

- Baseline set of ATF tests written for the original static array implementation

Generic ATF Runs

- Baseline set of ATF tests written for the original static array implementation
- `rbtree(3)` implementation would work as long as the baseline ATF Tests passed.

Generic ATF Runs

- Baseline set of ATF tests written for the original static array implementation
- `rbtree(3)` implementation would work as long as the baseline ATF Tests passed.
- 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

Generic ATF Runs

- Baseline set of ATF tests written for the original static array implementation
- `rbtree(3)` implementation would work as long as the baseline ATF Tests passed.
- 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
- However, there were some interesting “Edge Cases”

Case 1: `uvm_page_physload()`'s Prototype

- Function was originally designed to plug in segments of memory range during boot time.
- If any errors happened it would generally print a message and / or panic
- It was fine for `uvm_page_physload()` to return `void` after its execution in this scenario

Case 1: `uvm_page_physload()`'s Prototype

- Function was originally designed to plug in segments of memory range during boot time.
- If any errors happened it would generally print a message and / or panic
- It was fine for `uvm_page_physload()` to return `void` after its execution in this scenario
- But this was **NOT FINE** for the ATF Testing

Case 1: `uvm_page_physload()`'s Prototype

So what did we do?

Case 1: `uvm_page_physload()`'s Prototype

So what did we do?

We added a return value of type `uvm_physmem_t`

Case 1: `uvm_page_physload()`'s Prototype

So what did we do?

We added a return value of type `uvm_physmem_t`

Old Prototype

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

Case 1: `uvm_page_physload()`'s Prototype

So what did we do?

We added a return value of type `uvm_physmem_t`

Old Prototype

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

New Prototype

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

Case 1: `uvm_page_physload()`'s Prototype

So what did we do?

We added a return value of type `uvm_physmem_t`

Old Prototype

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

New Prototype

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

The tests became more concise, more readable and had unwanted assumptions removed from within.

Case 2: Immutable handles

- A particular test case `uvm_physseg_get_prev` kept failing for static array implementation but **not** R-B Tree implementation
- For the static array implementation we were using the `VM_PSTRAT_BSEARCH` strategy

Case 2: Immutable handles

- A particular test case `uvm_physseg_get_prev` kept failing for static array implementation but **not** R-B Tree implementation
- For the static array implementation we were using the `VM_PSTRAT_BSEARCH` strategy
- The test failed only if 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**

Case 2: Immutable handles

- A particular test case `uvm_physseg_get_prev` kept failing for static array implementation but **not** R-B Tree implementation
- For the static array implementation we were using the `VM_PSTRAT_BSEARCH` strategy
- The test failed only if 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**
- Consequence of changing the way the **handle of segment** was being referenced

Static array implementation

```
                    +-------+-------+-------+        +-------+-------+-------+
Segment  Info       |   B   |       |       |        |   A   |   B   |       |
                    +-----------------------+  +-->  +-----------------------+
Index               |   0   |   1   |   2   |        |   0   |   1   |   2   |
(uvm_physseg_t)     +-------+-------+-------+        +-------+-------+-------+
```

Case 2: Immutable handles

Static array implementation

```
                 +------+------+------+             +------+------+------+
Segment Info     |  B   |      |      |             |  A   |  B   |      |
                 +--------------------+    +-->     +--------------------+
Index            |  0   |  1   |  2   |             |  0   |  1   |  2   |
(uvm_physseg_t)  +------+------+------+             +------+------+------+
```

R-B Tree implementation

```
                +---+                                    +---+
                | B |                              +-----+ B |
                +---+            +-->              |     +---+
                                                   |
                                                   |
          +---+       +---+                      +-+-+         +---+
          |   |       |   |                      | A |         |   |
          +---+       +---+                      +---+         +---+
```

Note: The pointer to the nodes are the handles (`uvm_physseg_t`)

Case 2: Immutable handles

- In order to separately identify this property of mutability we added a new test case in ATF
  `uvm_physseg_handle_immutable`

Case 2: Immutable handles

- In order to separately identify this property of mutability we added a new test case in ATF `uvm_physseg_handle_immutable`
- This test is expected to **fail** for static array implementation

Case 2: Immutable handles

- In order to separately identify this property of mutability we added a new test case in ATF `uvm_physseg_handle_immutable`
- This test is expected to **fail** for static array implementation
- This test is expected to **pass** for R-B tree implementation

Case 2: Immutable handles

- In order to separately identify this property of mutability we added a new test case in ATF `uvm_physseg_handle_immutable`
- This test is expected to **fail** for static array implementation
- This test is expected to **pass** for R-B tree 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.

Booting the Kernel

Case 1: The init dance

The first boot resulted in a kernel **PANIC**

Case 1: The init dance

The first boot resulted in a kernel **PANIC**

- We quickly identified that `kmem(9)` is not available until `uvm_page_init()` has done with all the initialization

Case 1: The init dance

The first boot resulted in a kernel **PANIC**

- We quickly identified that `kmem(9)` is not available until `uvm_page_init()` has done with all the initialization
- Maintain a minimal “static array” whose size is `VM_PHYSSEG_MAX` and once the init process is over, switch over to the `kmem(9)` allocator
- `uvm.page_init_done` was used to distinguish when to switch over to `kmem(9)`

Case 1: The init dance

The first boot resulted in a kernel **PANIC**

- We quickly identified that `kmem(9)` is not available until `uvm_page_init()` has done with all the initialization
- Maintain a minimal “static array” whose size is `VM_PHYSSEG_MAX` and once the init process is over, switch over to the `kmem(9)` allocator
- `uvm.page_init_done` was used to distinguish when to switch over to `kmem(9)`
- We wrote wrappers for the `kmem(9)` allocators.
- `uvm_physseg_alloc()` and `uvm_physseg_free()`

Case 1: The init dance

The first boot resulted in a kernel **PANIC**

- We quickly identified that `kmem(9)` is not available until `uvm_page_init()` has done with all the initialization
- Maintain a minimal "static array" whose size is `VM_PHYSSEG_MAX` and once the init process is over, switch over to the `kmem(9)` allocator
- `uvm.page_init_done` was used to distinguish when to switch over to `kmem(9)`
- We wrote wrappers for the `kmem(9)` allocators.
- `uvm_physseg_alloc()` and `uvm_physseg_free()`
- Wrote up the test cases for these first, allowing for a smooth implementation

Case 2: Fragmentation of segments

What exactly is “fragmentation of a segment”?

Case 2: Fragmentation of segments

What exactly is “fragmentation of a segment”?

The `pgs[]` is contained in a given segment, allocated by `kmem(9)` allocators

```
+--------------------------------------------------+
|                    Segment A                     |
+--------------------------------------------------+
```

Case 2: Fragmentation of segments

What exactly is “fragmentation of a segment”?

The `pgs[]` is contained in a given segment, allocated by `kmem(9)` allocators

```
+-------------------------------------------------------+
|                      Segment A                        |
+-------------------------------------------------------+
```

So what happens to `pgs[]` if we “unplug” a section?

```
+---------------------------------+---------------------+
|             Segment A           |     Segment B       |
+---------------------------------+---------------------+
```

Case 2: Fragmentation of segments

What exactly is "fragmentation of a segment"?

The `pgs[]` is contained in a given segment, allocated by `kmem(9)` allocators

```
+--------------------------------------------------------------+
|                         Segment A                            |
+--------------------------------------------------------------+
```

So what happens to `pgs[]` if we "unplug" a section?

```
+--------------------------------------+-----------------------+
|              Segment A               |      Segment B        |
+--------------------------------------+-----------------------+
```

What happens to `pgs[]` if we "unplug" from the middle?

```
+--------------------+--------------------+--------------------+
|     Segment A      |     Segment C      |     Segment B      |
+--------------------+--------------------+--------------------+
```

Case 2: Fragmentation of segments

How did we solve this?

Case 2: Fragmentation of segments

How did we solve this?

- Use the `extent(9)` memory manager to manage the `pgs[]` array

Case 2: Fragmentation of segments

How did we solve this?

- Use the `extent(9)` memory manager to manage the `pgs[]` array
- We applied the “init dance” technique to solve Boot time vs non-Boot time allocation of slabs

Case 2: Fragmentation of segments

How did we solve this?

- Use the `extent(9)` memory manager to manage the `pgs[]` array
- We applied the “init dance” technique to solve Boot time vs non-Boot time allocation of slabs
- Once again extensive ATF tests that helped us out in minimising the downtime from debugging the code

Performance evaluation

Designing the test framework

...so we leveraged ATF to do this

- The most frequent opeation is `uvm_physseg_find()`

Designing the test framework

...so we leveraged ATF to do this

- The most frequent opeation is `uvm_physseg_find()`
- Copied over the `PHYS_TO_VM_PAGE()` macro and the related code from `uvm_page.c`

Designing the test framework

...so we leveraged ATF to do this

- The most frequent opeation is `uvm_physseg_find()`
- Copied over the `PHYS_TO_VM_PAGE()` macro and the related code from `uvm_page.c`
- Plug in segments and then do multiple calls to `PHYS_TO_VM_PAGE()`

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

Designing the test framework

...so we leveraged ATF to do this

- The most frequent opeation is `uvm_physseg_find()`
- Copied over the `PHYS_TO_VM_PAGE()` macro and the related code from `uvm_page.c`
- Plug in segments and then do multiple calls to `PHYS_TO_VM_PAGE()`

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

- After some tweaking around we managed to write up the tests varying from 100 calls to 100 Million calls

Designing the test framework

Things to Note

- This methodology is not a perfect load test since there is a call to `random()`
- This will cumulatively add up to the runtime of the function we are trying to load test.

Designing the test framework

Things to Note

- This methodology is not a perfect load test since there is a call to `random()`
- This will cumulatively add up to the runtime of the function we are trying to load test.
- All of the ATF tests have `ATF_CHECK_EQ(true, true)` at the bottom of the test indicating the test will never fail
- This is done because the test is **NOT** a check of correctness

Designing the test framework

We implemented two types of test strategies

- **Fixed size segment**: Here we plug in a “fixed” size segment. And pick a random address to do the `PHYS_TO_VM_PAGE()`. The variable here was the amount of calls done to `PHYS_TO_VM_PAGE()`

Designing the test framework

We implemented two types of test strategies

- **Fixed size segment**: Here we plug in a “fixed” size segment. And pick a random address to do the `PHYS_TO_VM_PAGE()`. The variable here was the amount of calls done to `PHYS_TO_VM_PAGE()`
- **Fragmented segment**: Here we plug in a known size segment. After which we start unplugging areas of the memory. Then we pick a random address to do `PHYS_TO_VM_PAGE()`. Here the variable was the memory size meaning, the bigger memory segment the more fragmented it was.

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

Note: In the results 100 consecutive runs were done and then the average, minimum and maximum runtimes were calculated.

```
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.
```

Benchmark results

Calls to `PHYS_TO_VM_PAGE()`

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: 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: Static array implementation

Calls to `PHYS_TO_VM_PAGE()`

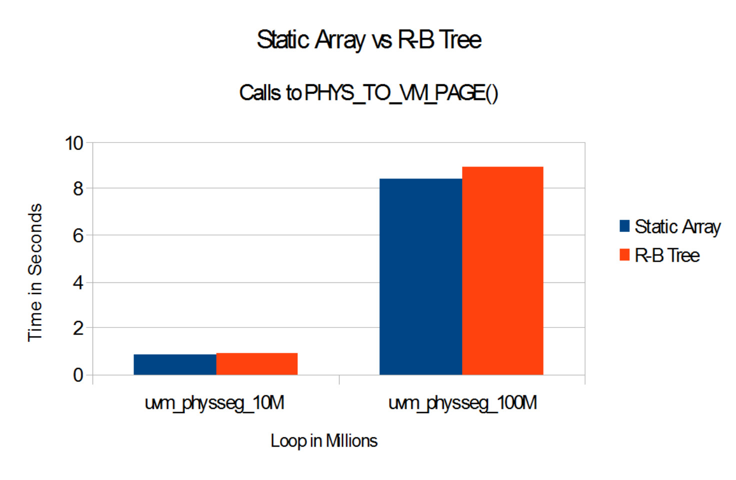


Figure 1: A closer look at the 10M and 100M calls side-by-side

Calls to `PHYS_TO_VM_PAGE()`

Since the 100M calls, took the most amount of time, we did some very specific analysis on this.

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()`

Calls to `PHYS_TO_VM_PAGE()`

Figure 2: Clearly there is a 5.59% degradation in performance with the R-B tree implementation

Calls to `PHYS_TO_VM_PAGE()` after fragmentation

- Number after test name indicates the amount of memory on which fragmentation was done
- Fragmentation was done by `uvm_physseg_unplug()`
- 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.

Calls to PHYS_TO_VM_PAGE() after fragmentation

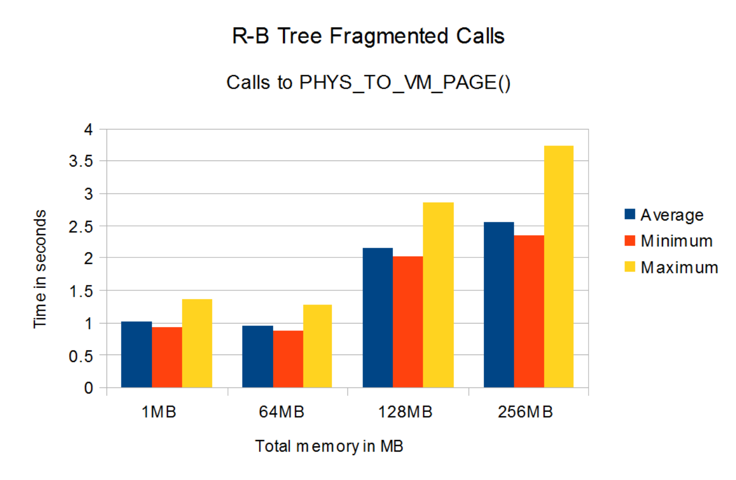


Figure 3: R-B tree performance for 10M Calls to `PHYS_TO_VM_PAGE()` after fragmentation at every 8 PFN

Conclusion and future work

Retrospective

Looking back...

- `rumpkernel(7)` based testing?
- Code coverage, maybe?
- Performance testing in an actual live kernel implementation with `dtrace(1)`

Conclusion

- Systems Programming can be made much less stressful by using existing Software Engineering techniques.

Conclusion

- Systems Programming can be made much less stressful by using existing Software Engineering techniques.
- The availability of general purpose APIs such as `rbtree(3)` and `extent(9)` in the NetBSD kernel, which makes implementation much less headache.

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.

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.

Credits and References

Thank you

- **The NetBSD Foundation** <http://www.NetBSD.org/foundation> generously funded this work.
- **KeK** <hello@kek.org.in> provided a cozy space right next to **Kovalam Beach** for us to hammer out the implementation.
- **Chuck Silvers** <chs@NetBSD.org> reviewed and helped refine the APIs. He also provided deep insight into the challenges of architecting such low level code.
- **Matthew Green** <mrg@NetBSD.org> made many helpful suggestions and critical feedback during the development and integration timeframe.
- **Maya Rashish** <maya@NetBSD.org> helped expose the API to multiple usecase situations (including header breakage in pkgsrc).
- **Nick Hudson** <skrll@NetBSD.org> contributed bugfixes, testing and integration on a wide range of hardware ports.
- **Philip Paeps** <philip@FreeBSD.org> helped guide creation, review and correction of the content of abstract and paper for `uvm_hotplug(9)`
- **Thomas Klausner** <wiz@NetBSD.org> helped make corrections to man page of `uvm_hotplug(9)`
- **Tom Flavel** <tom@printf.net> coerced cherry@NetBSD.org towards TDD, who was able to interest Santhosh Raju in applying the method to kernel programming. This allegedly turned out to be a good thing eventually.

Thank you

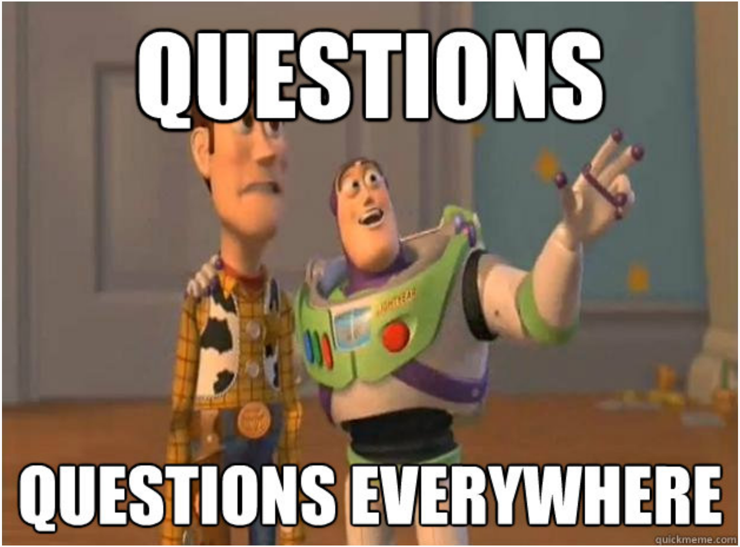
... And to all others who helped us along the way and we may have accidentally missed out or forgot to mention.

Thank you

... And to all others who helped us along the way and we may have accidentally missed out or forgot to mention.

And of course the **audience** for being here and patient while listening to the talk.

Questions



References

- `uvm_hotplug(9)` man page
  http://netbsd.gw.com/cgi-bin/man-cgi?uvm_hotplug++NetBSD-current
- `uvm_hotplug(9)` port-masters’ FAQ
  https://wiki.netbsd.org/features/uvm_hotplug/
- `uvm(9)` man page
  http://netbsd.gw.com/cgi-bin/man-cgi?uvm+9+NetBSD-current
- `rbtree(3)` man page
  http://netbsd.gw.com/cgi-bin/man-cgi?rbtree+3+NetBSD-current
- `atf(7)` man page
  http://netbsd.gw.com/cgi-bin/man-cgi?atf+7+NetBSD-current
- `uvm_hotplug(9)` development blog
  http://fraggerfox.homenet.org:10080/bsd-blog/

The End