Migration: One year later

KVM Forum 2011

Red Hat
Juan Quintela

August 15, 2011

Abstract

This talk describes current migration status, and ideas for future work.

Contents

1 What is the Current State

2 Things to do

3 Some solutions

Section 1

What is the Current State

redhat

What needs to be moved

- memory
  Have I told you that memory nowadays is big? Customer asking already for 8GB guests. Partners for 64-128GB guests.
- disk
  And you thought that memory was big. Think again.
- devices
  Size don't matter here (insert joke)
  But state is spread through a file, not always in a nice place that is trivial to sent.

redhat

What needs to be moved

- memory
  Have I told you that memory nowadays is big? Customer asking already for 8GB guests. Partners for 64-128GB guests.
- disk
  And you thought that memory was big. Think again.
- devices
  Size don't matter here (insert joke)
  But state is spread through a file, not always in a nice place that is trivial to sent.

redhat

What needs to be moved

- memory
  Have I told you that memory nowadays is big? Customer asking already for 8GB guests. Partners for 64-128GB guests.
- disk
  And you thought that memory was big. Think again.
- devices
  Size don't matter here (insert joke)
  But state is spread through a file, not always in a nice place that is trivial to sent.

Backward/Forward compatibility

- Old to Old and New to New
  Should be no problem (ha).
- Old → New
  We are in the future, we know what Old sent, should be easy. (famous last words).
- New → Old
  We are the future, wanting to sent something to the past, and we want the past to understand it. Think NP-complete.
  But we try, of course.

Backward/Forward compatibility

- Old to Old and New to New
  Should be no problem (ha).
- Old → New
  We are in the future, we know what Old sent, should be easy. (famous last words).
- New → Old
  We are the future, wanting to sent something to the past, and we want the past to understand it. Think NP-complete.
  But we try, of course.

redhat

Backward/Forward compatibility

- Old to Old and New to New
  Should be no problem (ha).
- Old → New
  We are in the future, we know what Old sent, should be easy. (famous last words).
- New → Old
  We are the future, wanting to sent something to the past, and we want the past to understand it. Think NP-complete.
  But we try, of course.

Sections, Subsections, Versions

A.K.A. Head hurts ...

- Sections: each device has one.
- Subsections: They are optional. Source decides if they are needed or not.
- Version: Each section has a section number. When we add some fields to a section, we increase the version number, and they are not expected from older versions, but are sent from new versions.

redhat

Sections, Subsections, Versions

A.K.A. Head hurts ...

- Sections: each device has one.
- Subsections: They are optional. Source decides if they are needed or not.
- Version: Each section has a section number. When we add some fields to a section, we increase the version number, and they are not expected from older versions, but are sent from new versions.

Sections, Subsections, Versions

A.K.A. Head hurts ...

- Sections: each device has one.
- Subsections: They are optional. Source decides if they are needed or not.
- Version: Each section has a section number. When we add some fields to a section, we increase the version number, and they are not expected from older versions, but are sent from new versions.

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?
  - We can sent only minimal amount of information that is always needed
  - Sent rest of information only when it is used
  - Source sent a subsection when it knows that it is needed
  - Target never discards a subsection
  - If it don't understand it, it just fails migration.

redhat

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?
  - We can sent only minimal amount of information that is always needed
  - Sent rest of information only when it is used
  - Source sent a subsection when it knows that it is needed
  - Target never discards a subsection
  - If it don't understand it, it just fails migration

redhat

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?

redhat

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?
  - We can sent only minimal amount of information that is always needed
  - Sent rest of information only when it is used
  - Source sent a subsection when it knows that it is needed
  - Target never discards a subsection.
    If it don't understand it, it just fails migration.

redhat

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?
  - We can sent only minimal amount of information that is always needed
  - Sent rest of information only when it is used
  - Source sent a subsection when it knows that it is needed
  - Target never discards a subsection.
    If it don't understand it, it just fails migration.

redhat

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?
  - We can sent only minimal amount of information that is always needed
  - Sent rest of information only when it is used
  - Source sent a subsection when it knows that it is needed
  - Target never discards a subsection.
    If it don't understand it, it just fails migration.

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?
  - We can sent only minimal amount of information that is always needed
  - Sent rest of information only when it is used
  - Source sent a subsection when it knows that it is needed
  - Target never discards a subsection.
    If it don't understand it, it just fails migration.

redhat

Subsections

- Cure cancer
- Get World Peace
- End World Hunger, ...
- Big idea: Why sent everything?
  - We can sent only minimal amount of information that is always needed
  - Sent rest of information only when it is used
  - Source sent a subsection when it knows that it is needed
  - Target never discards a subsection.
    If it don't understand it, it just fails migration.

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat

Live Migration: When the fun starts

- Memory migration
  - it is big
  - when we fail: memory corruption
  - crash of the machine
- Disk migration
  - you thought memory was big
  - when we fail: disk corruption
  - data loss
  - Will not talk more about disk
- From a 10000 meters view, memory and disk migration are equivalent

redhat.

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don’t you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don't you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don't you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don’t you like the meaning overload

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to "dirty" (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is "low enough" (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don't you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don’t you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to "dirty" (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is "low enough" (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don't you like the meaning overload?

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
  - Don't you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1

  Don't you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1

  Don't you like the meaning overload

redhat.

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to "dirty" (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is "low enough" (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1

  Don't you like the meaning overload

redhat.

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1

  Don’t you like the meaning overload

redhat

Live Migration: how it works?

- We have a dirty bitmap with one bit for each page
- We set all the bitmap to “dirty” (A)
- We loop through the bitmap: (B)
  - copy the page
  - clear the bit
- We end the loop when the number of dirty pages is “low enough” (B)
- We stop the machine (C)
- We sent the rest of the pages and all devices (C)
- Stages? What is that?
  - A: stage 1
  - B: stage 2
  - C: stage 3
  - cancel/error: stage -1
    Don’t you like the meaning overload

How qemu works?

A.K.A. Why we need threads for migration

- IOthread

```
....
while(1) {
    ....
    qemu_mutex_unlock_iothread();
    select(...)
    qemu_mutex_lock_iothread();
    .... /* We will refer to this part on the next slide */
}
```

- VCPU's

```
int kvm_cpu_exec(...)
{
    ...
    do {
        ....
        qemu_mutex_unlock_iothread();
        kvm_vcpu_ioctl(..)
        qemu_mutex_lock_iothread()
        ....
```

redhat

How qemu works?

A.K.A. Why we need threads for migration

- IOthread

```
....
while(1) {
    ....
    qemu_mutex_unlock_iothread();
    select(...)
    qemu_mutex_lock_iothread();
    .... /* We will refer to this part on the next slide */
}
```

- VCPU's

```
int  kvm_cpu_exec(...)
{
    ...
    do {
        ....
        qemu_mutex_unlock_iothread();
        kvm_vcpu_ioctl(..)
        qemu_mutex_lock_iothread()
        ....
```

redhat

What else iothread does?

```
...
QLIST_FOREACH_SAFE(ioh, &io_handlers, next, pioh) {
    if (...FD_ISSET(ioh->fd, readfs),...)
        ioh->fd_read(ioh->opaque)
    if (...FD_ISSET(ioh->fd, readfs),...)
        ioh->fd_write(ioh->opaque)
qemu_run_all_timers()
qemu_bh_poll()
```

How can this ever work?

- Don't this mean that things get "monothread"
- In general no, because
    - iohandlers run very fast
    - vcpu threads are out of guest very few times
    - Rest of things cheat
    - migration: where the abstraction leaks

redhat

How can this ever work?

- Don't this mean that things get "monothread"
- In general no, because
  - iohandlers run very fast
  - vcpu threads are out of guest very few times
  - Rest of things cheat
    - block layer: async IO
    - networking: vhost + async IO
  - migration: where the abstraction leaks

How can this ever work?

- Don't this mean that things get "monothread"
- In general no, because
  - iohandlers run very fast
  - vcpu threads are out of guest very few times
  - Rest of things cheat
    - block layer: async IO
    - networking: vhost + async IO
  - migration: where the abstraction leaks

How can this ever work?

- Don’t this mean that things get “monothread”
- In general no, because
  - iohandlers run very fast
  - vcpu threads are out of guest very few times
  - Rest of things cheat
    - block layer: async IO
    - networking: vhost + async IO
  - migration: where the abstraction leaks

redhat

How can this ever work?

- Don't this mean that things get "monothread"
- In general no, because
  - iohandlers run very fast
  - vcpu threads are out of guest very few times
  - Rest of things cheat
    - block layer: async IO
    - networking: vhost + async IO
  - migration: where the abstraction leaks

redhat

How can this ever work?

- Don't this mean that things get "monothread"
- In general no, because
  - iohandlers run very fast
  - vcpu threads are out of guest very few times
  - Rest of things cheat
    - block layer: async IO
    - networking: vhost + async IO
  - migration: where the abstraction leaks

redhat

How can this ever work?

- Don't this mean that things get "monothread"
- In general no, because
  - iohandlers run very fast
  - vcpu threads are out of guest very few times
  - Rest of things cheat
    - block layer: async IO
    - networking: vhost + async IO
  - migration: where the abstraction leaks

redhat

How can this ever work?

- Don't this mean that things get "monothread"
- In general no, because
  - iohandlers run very fast
  - vcpu threads are out of guest very few times
  - Rest of things cheat
    - block layer: async IO
    - networking: vhost + async IO
  - migration: where the abstraction leaks

redhat

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

redhat

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

redhat

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

redhat

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

redhat

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

redhat

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

Buffered file

A.K.A. Another buffer layer will fix any computing problem

- Migration runs in an IOHandler
- But it can't stop in the middle of a device
- We add an autogrowing buffer to be able to always finish device state write
- And we write with a timer that buffer to a FILE *
- We wait with select in the FILE * descriptor
- We write it with write()
- And Kernel wants to do its own buffering
- Enough buffering for you?

Measurements: who needs that?

- We have two knobs
  - migrate_speed: in MB
    Yes, I mean that, we measure speed in Megabytes, think about it.
  - max_downtime: in ms
- And we try to make sense of them.
- When migration don't converge, we don't know for how much

redhat

Measurements: who needs that?

- We have two knobs
  - migrate_speed: in MB
    Yes, I mean that, we measure speed in Megabytes, think about it.
  - max_downtime: in ms
- And we try to make sense of them.
- When migration don't converge, we don't know for how much

Measurements: who needs that?

- We have two knobs
  - migrate_speed: in MB
    Yes, I mean that, we measure speed in Megabytes, think about it.
  - max_downtime: in ms
- And we try to make sense of them.
- When migration don't converge, we don't know for how much

Measurements: who needs that?

- We have two knobs
  - migrate_speed: in MB
    Yes, I mean that, we measure speed in Megabytes, think about it.
  - max_downtime: in ms
- And we try to make sense of them.
- When migration don't converge, we don't know for how much

redhat

Measurements: who needs that?

- We have two knobs
  - migrate_speed: in MB
    Yes, I mean that, we measure speed in Megabytes, think about it.
  - max_downtime: in ms
- And we try to make sense of them.
- When migration don't converge, we don't know for how much

redhat

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?
  - We are measuring how fast we can write to a FILE * buffer
  - We don't measure how big/fast/loaded is the network
  - We have a nice optimization that sent a byte for each page
  - If we have lots of blank pages we spent a lot of time to sent them

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?
  - We are measuring how fast we can write to a FILE * buffer
  - We don't measure how big/fast/loaded is the network
  - We have a nice optimization that sent a byte for each page
  - If we have lots of blank pages we spent a lot of time to sent them

redhat.

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?

redhat

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?
  - We are measuring how fast we can write to a FILE * buffer
  - We don't measure how big/fast/loaded is the network
  - We have a **nice** optimization that sent a byte for each page
  - If we have lots of blank pages we spent a lot of time to sent them

redhat

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?
  - We are measuring how fast we can write to a FILE * buffer
  - We don't measure how big/fast/loaded is the network
  - We have a **nice** optimization that sent a byte for each page
  - If we have lots of blank pages we spent a lot of time to sent them

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?
  - We are measuring how fast we can write to a FILE * buffer
  - We don't measure how big/fast/loaded is the network
  - We have a **nice** optimization that sent a byte for each page
  - If we have lots of blank pages we spent a lot of time to sent them

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?
  - We are measuring how fast we can write to a FILE * buffer
  - We don't measure how big/fast/loaded is the network
  - We have a **nice** optimization that sent a byte for each page
  - If we have lots of blank pages we spent a lot of time to sent them

migration speed

- Remember the buffered file
- Remember that we measure speed in megabytes?
- migration handler interesting part is:

```
while (number_bytes_sent < max_speed) {
  sent_another_page()
}
```

- What can be wrong with this?
  - We are measuring how fast we can write to a FILE * buffer
  - We don't measure how big/fast/loaded is the network
  - We have a **nice** optimization that sent a byte for each page
  - If we have lots of blank pages we spent a lot of time to sent them

redhat

Incoming migration

A.K.A Who needs a toplevel while we do incoming migration

- We don't have toplevel
- Libvirt/user can't ask **anything**
- Everything had to be configured from the command line
- Cancellation can only happen on the outgoing migration side

redhat

Incoming migration

A.K.A Who needs a toplevel while we do incoming migration

- We don't have toplevel
- Libvirt/user can't ask **anything**
- Everything had to be configured from the command line
- Cancellation can only happen on the outgoing migration side

redhat

Incoming migration

A.K.A Who needs a toplevel while we do incoming migration

- We don't have toplevel
- Libvirt/user can't ask **anything**
- Everything had to be configured from the command line
- Cancellation can only happen on the outgoing migration side

redhat.

Incoming migration

A.K.A Who needs a toplevel while we do incoming migration

- We don't have toplevel
- Libvirt/user can't ask **anything**
- Everything had to be configured from the command line
- Cancellation can only happen on the outgoing migration side

Section 2

Things to do

redhat

VMState: Finish the work

- Virtio devices: old code exists. Problem is that we have list of requests, and we have no good idea how to represent lists on VMState.
- Rest of CPU's: no real problem, just code that needs to be written. (sections are quite big).
- slirp: eats puppies. Slirp code is a mess, It is lists of lists of lists. Code needs fixing independently of VMState.
- Rest of misc devices: Ugliness:
  - bitfields fields
  - size differences between vmstate and state
  - other misc things

redhat

VMState: Finish the work

- Virtio devices: old code exists. Problem is that we have list of requests, and we have no good idea how to represent lists on VMState.
- Rest of CPU's: no real problem, just code that needs to be written. (sections are quite big).
- slirp: eats puppies. Slirp code is a mess, It is lists of lists of lists. Code needs fixing independently of VMState.
- Rest of misc devices: Ugliness:
    - bitfields fields
    - size differences between vmstate and state
    - other misc things

redhat

VMState: Finish the work

- Virtio devices: old code exists. Problem is that we have list of requests, and we have no good idea how to represent lists on VMState.
- Rest of CPU's: no real problem, just code that needs to be written. (sections are quite big).
- slirp: eats puppies. Slirp code is a mess, It is lists of lists of lists. Code needs fixing independently of VMState.
- Rest of misc devices: Ugliness:
  - bitfields fields
  - size differences between vmstate and state
  - other misc things

redhat

VMState: Finish the work

- Virtio devices: old code exists. Problem is that we have list of requests, and we have no good idea how to represent lists on VMState.
- Rest of CPU's: no real problem, just code that needs to be written. (sections are quite big).
- slirp: eats puppies. Slirp code is a mess, It is lists of lists of lists. Code needs fixing independently of VMState.
- Rest of misc devices: Ugliness:
  - bitfields fields
  - size differences between vmstate and state
  - other misc things

redhat

VMState: Finish the work

- Virtio devices: old code exists. Problem is that we have list of requests, and we have no good idea how to represent lists on VMState.
- Rest of CPU's: no real problem, just code that needs to be written. (sections are quite big).
- slirp: eats puppies. Slirp code is a mess, It is lists of lists of lists. Code needs fixing independently of VMState.
- Rest of misc devices: Ugliness:
  - bitfields fields
  - size differences between vmstate and state
  - other misc things

VMState: Finish the work

- Virtio devices: old code exists. Problem is that we have list of requests, and we have no good idea how to represent lists on VMState.
- Rest of CPU's: no real problem, just code that needs to be written. (sections are quite big).
- slirp: eats puppies. Slirp code is a mess, It is lists of lists of lists. Code needs fixing independently of VMState.
- Rest of misc devices: Ugliness:
  - bitfields fields
  - size differences between vmstate and state
  - other misc things

redhat

VMState: Finish the work

- Virtio devices: old code exists. Problem is that we have list of requests, and we have no good idea how to represent lists on VMState.
- Rest of CPU's: no real problem, just code that needs to be written. (sections are quite big).
- slirp: eats puppies. Slirp code is a mess, It is lists of lists of lists. Code needs fixing independently of VMState.
- Rest of misc devices: Ugliness:
  - bitfields fields
  - size differences between vmstate and state
  - other misc things

redhat

Subsections

- Detection of subsection is wrong, only looks at the 1st byte
- Needs to look at the whole header, and see if len + name makes sense
- It requires the equivalent of ungetc() to work for 10-20 chars. And it has to work in the middle of two packets.
- Needs to be done, details and testing are the problem.
- mail with suggestions sent to qemu-devel@.

Subsections

- Detection of subsection is wrong, only looks at the 1st byte
- Needs to look at the whole header, and see if len + name makes sense
- It requires the equivalent of ungetc() to work for 10-20 chars. And it has to work in the middle of two packets.
- Needs to be done, details and testing are the problem.
- mail with suggestions sent to qemu-devel@.

Subsections

- Detection of subsection is wrong, only looks at the 1st byte
- Needs to look at the whole header, and see if len + name makes sense
- It requires the equivalent of ungetc() to work for 10-20 chars. And it has to work in the middle of two packets.
- Needs to be done, details and testing are the problem.
- mail with suggestions sent to qemu-devel@.

redhat

Subsections

- Detection of subsection is wrong, only looks at the 1st byte
- Needs to look at the whole header, and see if len + name makes sense
- It requires the equivalent of ungetc() to work for 10-20 chars. And it has to work in the middle of two packets.
- Needs to be done, details and testing are the problem.
- mail with suggestions sent to qemu-devel@.

redhat

Subsections

- Detection of subsection is wrong, only looks at the 1st byte
- Needs to look at the whole header, and see if len + name makes sense
- It requires the equivalent of ungetc() to work for 10-20 chars. And it has to work in the middle of two packets.
- Needs to be done, details and testing are the problem.
- mail with suggestions sent to qemu-devel@.

Migration Thread outgoing

A.K.A. Fix World Problems at once

- stalls on the vcpu/iohandler: gone
- buffered file: gone
- we can now measure better what is the speed that we are sending/receiving
- saturate networking: we are our own thread, blocking is ok
- Problem: how to handle dirty bitmap

Migration Thread outgoing

A.K.A. Fix World Problems at once

- stalls on the vcpu/iohandler: gone
- buffered file: gone
- we can now measure better what is the speed that we are sending/receiving
- saturate networking: we are our own thread, blocking is ok
- Problem: how to handle dirty bitmap

redhat

Migration Thread outgoing

A.K.A. Fix World Problems at once

- stalls on the vcpu/iohandler: gone
- buffered file: gone
- we can now measure better what is the speed that we are sending/receiving
- saturate networking: we are our own thread, blocking is ok
- Problem: how to handle dirty bitmap

Migration Thread outgoing

A.K.A. Fix World Problems at once

- stalls on the vcpu/iohandler: gone
- buffered file: gone
- we can now measure better what is the speed that we are sending/receiving
- saturate networking: we are our own thread, blocking is ok
- Problem: how to handle dirty bitmap

redhat

Migration Thread outgoing

A.K.A. Fix World Problems at once

- stalls on the vcpu/iohandler: gone
- buffered file: gone
- we can now measure better what is the speed that we are sending/receiving
- saturate networking: we are our own thread, blocking is ok
- Problem: how to handle dirty bitmap

Incoming Migration Thread

- we can have a toplevel back on incoming side
- Everything can works as usual, from the monitor
- IOThread is running, we can use it

redhat

Incoming Migration Thread

- we can have a toplevel back on incoming side
- Everything can works as usual, from the monitor
- IOThread is running, we can use it

redhat

Incoming Migration Thread

- we can have a toplevel back on incoming side
- Everything can works as usual, from the monitor
- IOThread is running, we can use it

redhat

Dirty Bitmap

A.K.A. What is that?

- Dirty bitmap has 8 bits for each page. CODE, VGA, MIGRATION
- move to 3 bitmaps: 70 percent size reduction
- who produces dirty pages: kvm, mmio
- who consumes dirty pages: vga, code, migration
- add avi, shake well, and .... **idea**
- use one bitmap for producer, and consumer syncs bitmaps each time it needs it
- this makes it almost thread safe by design

Dirty Bitmap

A.K.A. What is that?

- Dirty bitmap has 8 bits for each page. CODE, VGA, MIGRATION
- move to 3 bitmaps: 70 percent size reduction
- who produces dirty pages: kvm, mmio
- who consumes dirty pages: vga, code, migration
- add avi, shake well, and .... **idea**
- use one bitmap for producer, and consumer syncs bitmaps each time it needs it
- this makes it almost thread safe by design

redhat

Dirty Bitmap

A.K.A. What is that?

- Dirty bitmap has 8 bits for each page. CODE, VGA, MIGRATION
- move to 3 bitmaps: 70 percent size reduction
- who produces dirty pages: kvm, mmio
- who consumes dirty pages: vga, code, migration
- add avi, shake well, and .... **idea**
- use one bitmap for producer, and consumer syncs bitmaps each time it needs it
- this makes it almost thread safe by design

redhat

Dirty Bitmap

A.K.A. What is that?

- Dirty bitmap has 8 bits for each page. CODE, VGA, MIGRATION
- move to 3 bitmaps: 70 percent size reduction
- who produces dirty pages: kvm, mmio
- who consumes dirty pages: vga, code, migration
- add avi, shake well, and .... **idea**
- use one bitmap for producer, and consumer syncs bitmaps each time it needs it
- this makes it almost thread safe by design

redhat

Dirty Bitmap

A.K.A. What is that?

- Dirty bitmap has 8 bits for each page. CODE, VGA, MIGRATION
- move to 3 bitmaps: 70 percent size reduction
- who produces dirty pages: kvm, mmio
- who consumes dirty pages: vga, code, migration
- add avi, shake well, and .... **idea**
- use one bitmap for producer, and consumer syncs bitmaps each time it needs it
- this makes it almost thread safe by design

redhat

Dirty Bitmap

A.K.A. What is that?

- Dirty bitmap has 8 bits for each page. CODE, VGA, MIGRATION
- move to 3 bitmaps: 70 percent size reduction
- who produces dirty pages: kvm, mmio
- who consumes dirty pages: vga, code, migration
- add avi, shake well, and .... **idea**
- use one bitmap for producer, and consumer syncs bitmaps each time it needs it
- this makes it almost thread safe by design

redhat

Dirty Bitmap

A.K.A. What is that?

- Dirty bitmap has 8 bits for each page. CODE, VGA, MIGRATION
- move to 3 bitmaps: 70 percent size reduction
- who produces dirty pages: kvm, mmio
- who consumes dirty pages: vga, code, migration
- add avi, shake well, and .... **idea**
- use one bitmap for producer, and consumer syncs bitmaps each time it needs it
- this makes it almost thread safe by design

redhat

Dirty Bitmap II

A.K.A. More size reduction

- We have a ram list of ramblocks
- And a dirty bitmap from address 0 to max allocated address
- So, we have bitmap for holes (not needed)
- solution: move bitmap to ramblock instead of ramlist
- but you need to fix all exec.c users (TCG a.k.a. ugly)
- Why all operations are on guest addresses instead of ramblocks

redhat

Dirty Bitmap II

A.K.A. More size reduction

- We have a ram list of ramblocks
- And a dirty bitmap from address 0 to max allocated address
- So, we have bitmap for holes (not needed)
- solution: move bitmap to ramblock instead of ramlist
- but you need to fix all exec.c users (TCG a.k.a. ugly)
- Why all operations are on guest addresses instead of ramblocks

redhat

Dirty Bitmap II

A.K.A. More size reduction

- We have a ram list of ramblocks
- And a dirty bitmap from address 0 to max allocated address
- So, we have bitmap for holes (not needed)
- solution: move bitmap to ramblock instead of ramlist
- but you need to fix all exec.c users (TCG a.k.a. ugly)
- Why all operations are on guest addresses instead of ramblocks

redhat

Dirty Bitmap II

A.K.A. More size reduction

- We have a ram list of ramblocks
- And a dirty bitmap from address 0 to max allocated address
- So, we have bitmap for holes (not needed)
- solution: move bitmap to ramblock instead of ramlist
- but you need to fix all exec.c users (TCG a.k.a. ugly)
- Why all operations are on guest addresses instead of ramblocks

redhat

Dirty Bitmap II

A.K.A. More size reduction

- We have a ram list of ramblocks
- And a dirty bitmap from address 0 to max allocated address
- So, we have bitmap for holes (not needed)
- solution: move bitmap to ramblock instead of ramlist
- but you need to fix all exec.c users (TCG a.k.a. ugly)
- Why all operations are on guest addresses instead of ramblocks

redhat

Dirty Bitmap II

A.K.A. More size reduction

- We have a ram list of ramblocks
- And a dirty bitmap from address 0 to max allocated address
- So, we have bitmap for holes (not needed)
- solution: move bitmap to ramblock instead of ramlist
- but you need to fix all exec.c users (TCG a.k.a. ugly)
- Why all operations are on guest addresses instead of ramblocks

redhat

Migration protocol format

A.K.A. The more ugly

- Since Fortran60 everybody knows that you need a begin/end to mark zones/sections
- Qemu hasn't learned it, so there is no way to handle the format from the outside qemu
- Solution start/end markers + size
- checksums: cpu is cheap

redhat

Migration protocol format

A.K.A. The more ugly

- Since Fortran60 everybody knows that you need a begin/end to mark zones/sections
- Qemu hasn't learned it, so there is no way to handle the format from the outside qemu
- Solution start/end markers + size
- checksums: cpu is cheap

Migration protocol format

A.K.A. The more ugly

- Since Fortran60 everybody knows that you need a begin/end to mark zones/sections
- Qemu hasn't learned it, so there is no way to handle the format from the outside qemu
- Solution start/end markers + size
- checksums: cpu is cheap

Migration protocol format

A.K.A. The more ugly

- Since Fortran60 everybody knows that you need a begin/end to mark zones/sections
- Qemu hasn't learned it, so there is no way to handle the format from the outside qemu
- Solution start/end markers + size
- checksums: cpu is cheap

redhat

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat.

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat.

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat.

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat

Backward migration

A.K.A. Qdev is incomplete

- `-M pc-0.14` lies, and uses the same devices that v14
- but it uses the versions of v0.15.
- We need a way to tell a device: boot with version foo
- or without features foo+bar
- And then we can use that for migration.
- People continue asking that we fix that at migration level, but solution needs to be at qdev level. Otherwise, we are trying to boot a device with feature foo, and now magically, migration have to migration **without** feature foo.
- And get it working.
- Almost nobody care about backward migration
- But big cpu farms custormes care

redhat.

Section 3

Some solutions

Change the migration format

- Suggestion: move to ASN.1
- Suggestion: move to XML
- .....
- That helps describing the data in the wire, but helps with the other problems how?

Change the migration format

- Suggestion: move to ASN.1
- Suggestion: move to XML
- .....
- That helps describing the data in the wire, but helps with the other problems how?

redhat

Change the migration format

- Suggestion: move to ASN.1
- Suggestion: move to XML
- .....
- That helps describing the data in the wire, but helps with the other problems how?

redhat

Change the migration format

- Suggestion: move to ASN.1
- Suggestion: move to XML
- .....
- That helps describing the data in the wire, but helps with the other problems how?

redhat

All that needs to be changed is

```
static void put_int32(QEMUFile *f, void *pv, size_t size)
{
    int32_t *v = pv;
    qemu_put_sbe32s(f, v);
}
```

```
static void put_xml_int32(QEMUFile *f, void *pv, size_t size)
{
    int32_t *v = pv;
    printf("<value type=int32>%d</value>",*v);
}
```

All that needs to be changed is

```
static void put_int32(QEMUFile *f, void *pv, size_t size)
{
    int32_t *v = pv;
    qemu_put_sbe32s(f, v);
}
```

```
static void put_xml_int32(QEMUFile *f, void *pv, size_t size)
{
    int32_t *v = pv;
    printf("<value type=int32>%d</value>",*v);
}
```

redhat

One device gets split in 2 devices

A.K.A. Anthony, I am looking at you

```
struct OldState {
  int foo;
  int bar;
}
struct FooState {
  int foo;
}
struct BarState {
  int  bar;
}
```

redhat.

One device gets split in 2 devices (II)

```
struct OldState {
  int foo;
  int bar;
  struct FooState *foo;
}
struct FooState {
  int foo;
}
```

redhat

One device gets split in 2 devices (III)

```
static int old_state_post_load(void *opaque, int version_id)
{
    OldState *s = opaque;
    s->foo->foo = s->foo;
    return 0;
}

static const VMStateDescription vmstate_foo = {
    .name = "old_state",
    .post_load = old_state_post_load,
    .fields      = (VMStateField []) {
        VMSTATE_INT32(foo, OldState),
        VMSTATE_INT32(bar, OldState),
        VMSTATE_END_OF_LIST()
    }
}
```

redhat

Postcopy

- Networking vs CPU/RAM
- we have a new failure case
- but .... we only have to copy each page only once
- guest performance varies
- should be possible to do using current infrastructure

redhat

Postcopy

- Networking vs CPU/RAM
- we have a new failure case
- but .... we only have to copy each page only once
- guest performance varies
- should be possible to do using current infrastructure

redhat

Postcopy

- Networking vs CPU/RAM
- we have a new failure case
- but .... we only have to copy each page only once
- guest performance varies
- should be possible to do using current infrastructure

redhat

Postcopy

- Networking vs CPU/RAM
- we have a new failure case
- but .... we only have to copy each page only once
- guest performance varies
- should be possible to do using current infrastructure

redhat

Postcopy

- Networking vs CPU/RAM
- we have a new failure case
- but .... we only have to copy each page only once
- guest performance varies
- should be possible to do using current infrastructure

redhat

Conclusions

A.K.A. There is no conclusion of migration issues

- VMState: this needs to be finish
- On wire protocol: being/end/size/checksum?
- Migration thread: Umesh code good start
- Bitmap handling: something more reasonable
- measurements: we need more and better
- Post copy?

redhat

Conclusions

A.K.A. There is no conclusion of migration issues

- VMState: this needs to be finish
- On wire protocol: being/end/size/checksum?
- Migration thread: Umesh code good start
- Bitmap handling: something more reasonable
- measurements: we need more and better
- Post copy?

redhat

Conclusions

A.K.A. There is no conclusion of migration issues

- VMState: this needs to be finish
- On wire protocol: being/end/size/checksum?
- Migration thread: Umesh code good start
- Bitmap handling: something more reasonable
- measurements: we need more and better
- Post copy?

redhat

Conclusions

A.K.A. There is no conclusion of migration issues

- VMState: this needs to be finish
- On wire protocol: being/end/size/checksum?
- Migration thread: Umesh code good start
- Bitmap handling: something more reasonable
- measurements: we need more and better
- Post copy?

redhat

Conclusions

A.K.A. There is no conclusion of migration issues

- VMState: this needs to be finish
- On wire protocol: being/end/size/checksum?
- Migration thread: Umesh code good start
- Bitmap handling: something more reasonable
- measurements: we need more and better
- Post copy?

redhat

Conclusions

A.K.A. There is no conclusion of migration issues

- VMState: this needs to be finish
- On wire protocol: being/end/size/checksum?
- Migration thread: Umesh code good start
- Bitmap handling: something more reasonable
- measurements: we need more and better
- Post copy?

Questions?

The end.

Thanks for listening.