Inside Android's Dalvik VM

Fork Me on GitHub

Douglas Q. Hawkins
http://www.slideshare.net/dougqh
http://github.com/dougqh/nejug2011
http://www.dougqh.net
dougqh@gmail.com

Wednesday, November 9, 11

I'm Doug Hawkins -- interested in VMs -- spoke last year on JVM internals
Spent 6 months learning Dalvik
Ask questions as we go
In addition to being available on SlideShare, all research materials available on GitHub
Slides all have notes and citations for offline reading

Android Physiology - Patrick Brady

https://sites.google.com/site/io/anatomy--physiology-of-an-android

Dalvik VM Internals - Dan Bornstein

http://www.youtube.com/watch?v=ptjedOZEXPM

A JIT for Android's Dalvik VM

Ben Cheng and Bill Buzbee

http://www.youtube.com/watch?v=Ls0tM-c4Vfo

Based on Google IO presentations (mostly from 2008), but updated for the last 3 years

≠

DEX

JAR

Dalvik Bytecode Java Bytecode

Talking about Android

But more specifically talking about how Android and Java are different and why

And, how they are the same and why

Different Environments

528 MhZ ARM
192 MB RAM
100 MhZ Bus
32K Cache
No Swap!

http://www.youtube.com/watch?v=ptjedOZEXPM
http://www.gsmarena.com/t_mobile_g1-2533.php
Phone tech lags desktop tech by 10 years
Designed for as little as 70MB of RAM - only 10M left for apps

JVMs Problems

Memory Hog

Slow Startup

Talking about Android
But more specifically talking about how Android and Java are different and why
And, how they are the same and why

Not a 10 Year Old OS

http://en.wikipedia.org/wiki/Java_(programming_language)

Java was released in 1996, when Windows 95 was becoming the dominant OS

While an improvement over Windows 3.1, Windows 95 is not...

- secure
- multi-user

Old Mac OS was not event pre-emptive.

Dalvik gets a boost by depending on a modern OS - Linux

- and custom drivers

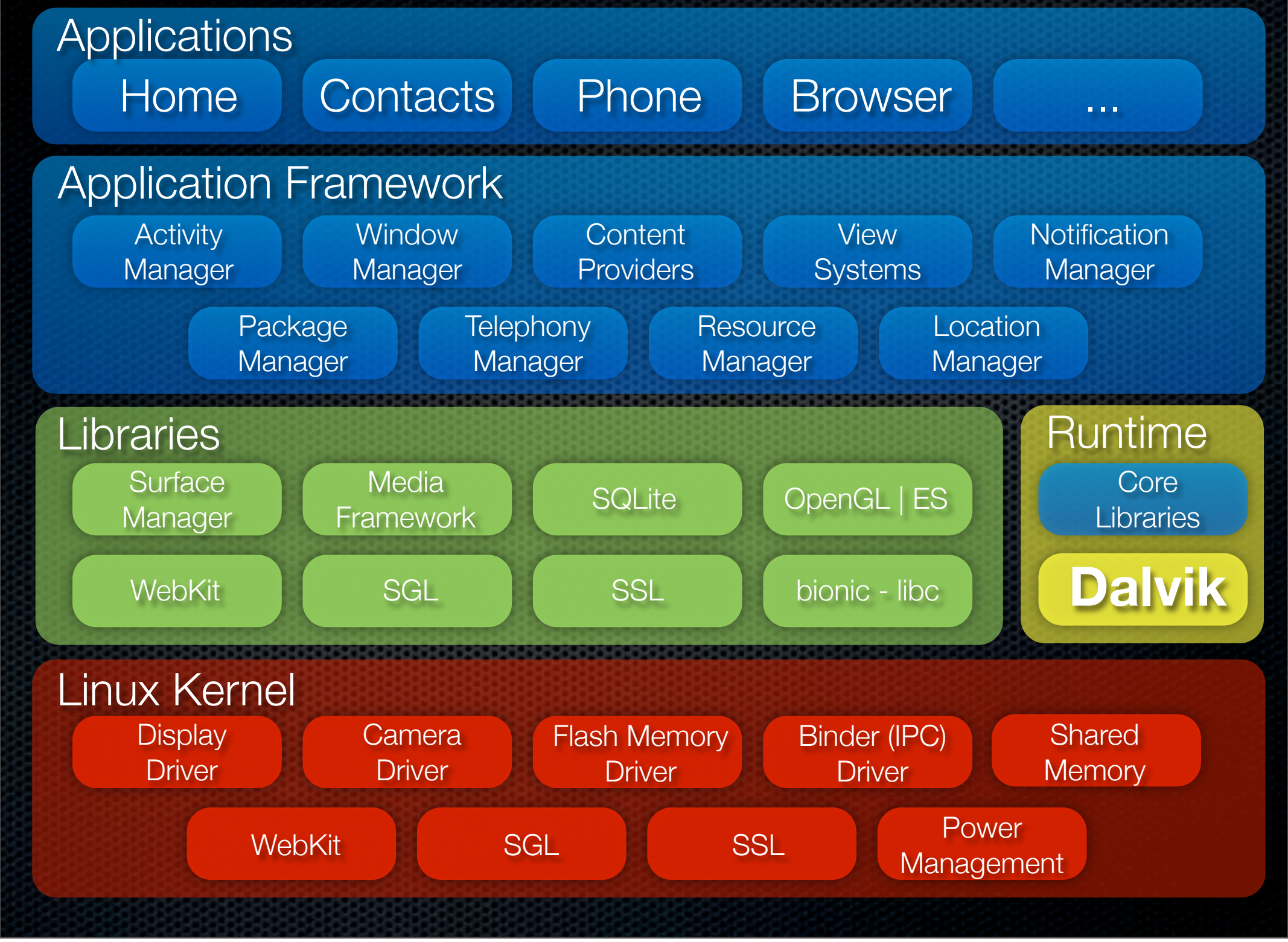
Dalvik

http://developer.android.com/guide/basics/what-is-android.html

Went into my research planning to just talk about the Dalvik VM...

...but found that Dalvik is just a piece of a larger puzzle...

...and the rest of the pieces influence Dalvik's design.



http://developer.android.com/guide/basics/what-is-android.html

Went into my research planning to just talk about the Dalvik VM...

...but found that Dalvik is just a piece of a larger puzzle...

...and the rest of the pieces influence Dalvik's design.

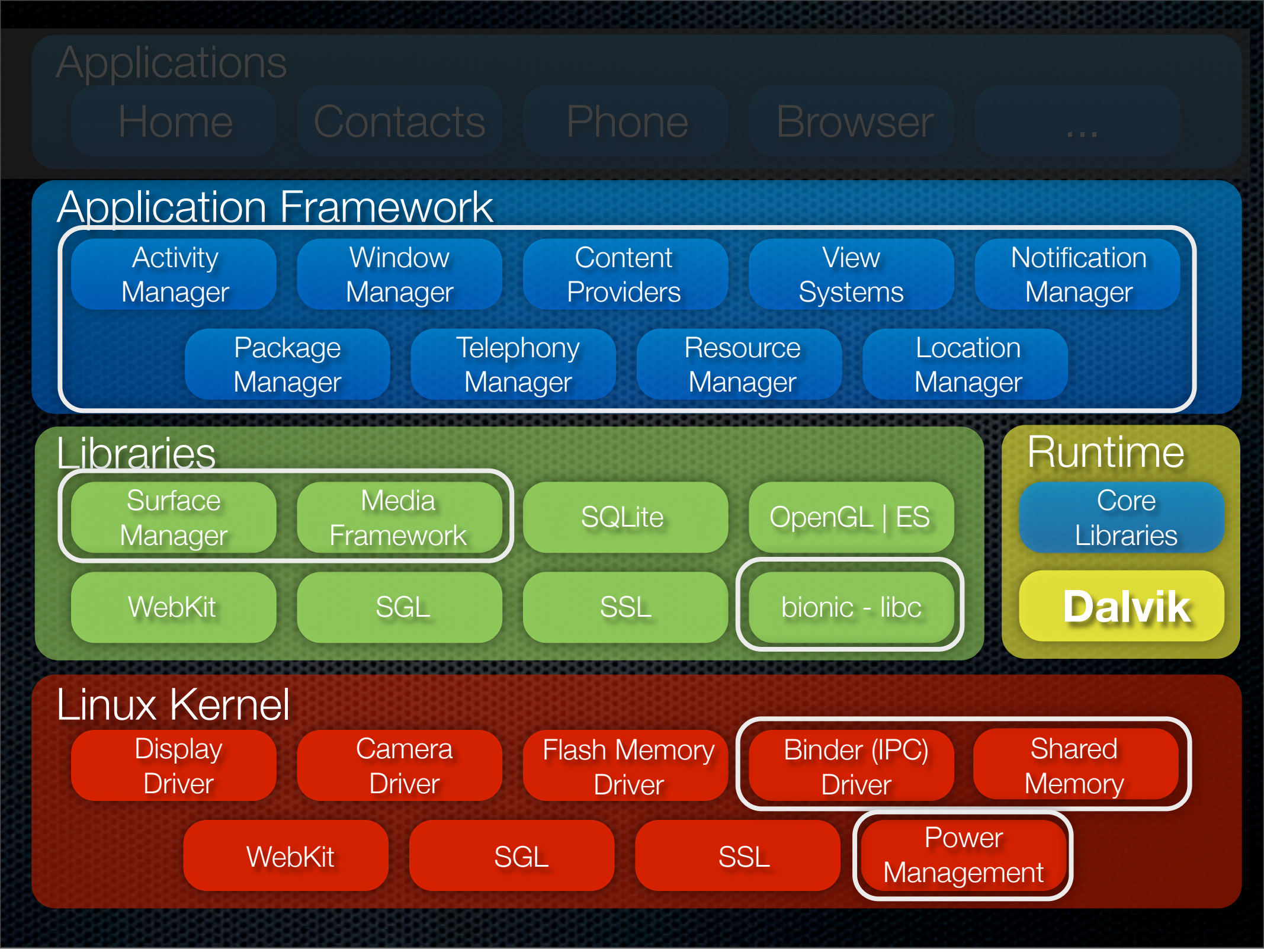
Legend

Java

Dalvik

Native

OS



So, the first part of this talk is about the parts of Android other than Dalvik -- except for applications --
specifically, Android's security and IPC mechanisms.

≠

Paranoid Networking
Power Management

Ashmem & Binder
Low Memory Killer

http://www.lindusembedded.com/blog/2010/12/07/android-linux-kernel-additions/
http://www.kroah.com/log/linux/android-kernel-problems.html

Power Management

Screen is off by default

CPU is off by default

“Wake” locks to stay on

http://developer.android.com/reference/android/os/PowerManager.html

Aggressive approach to power management - CPU and screen off by default

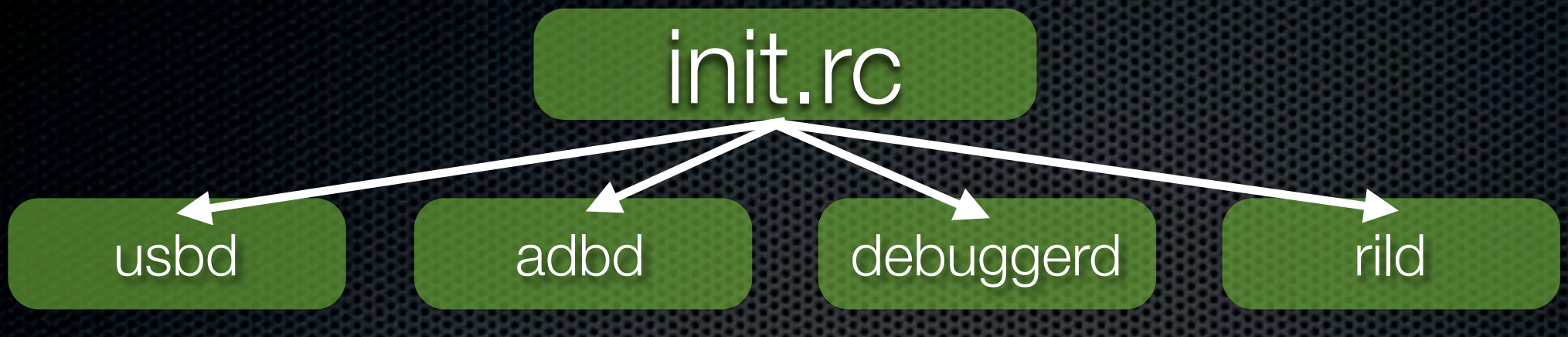
Telephony chip, etc. still responding

“Wake” locks to stay on

init.rc

https://sites.google.com/site/io/anatomy--physiology-of-an-android

usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home

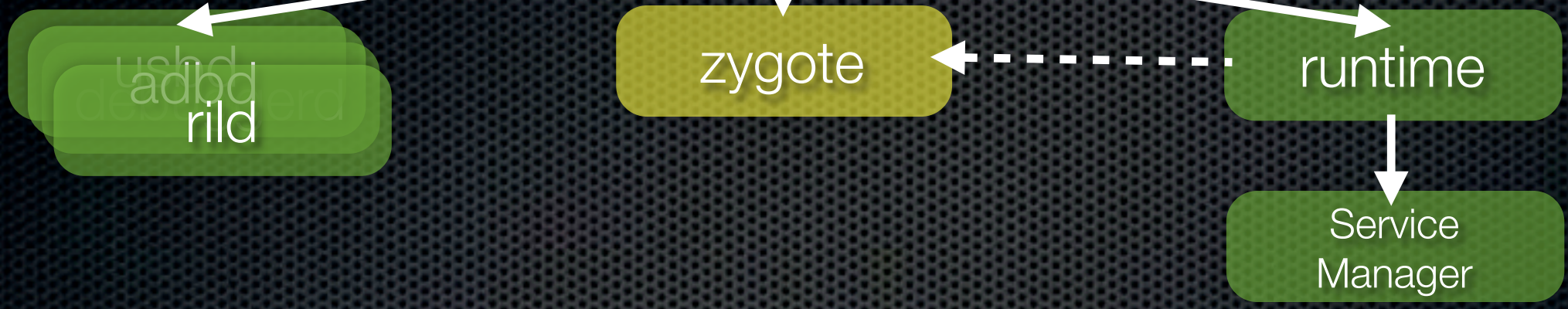


https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home

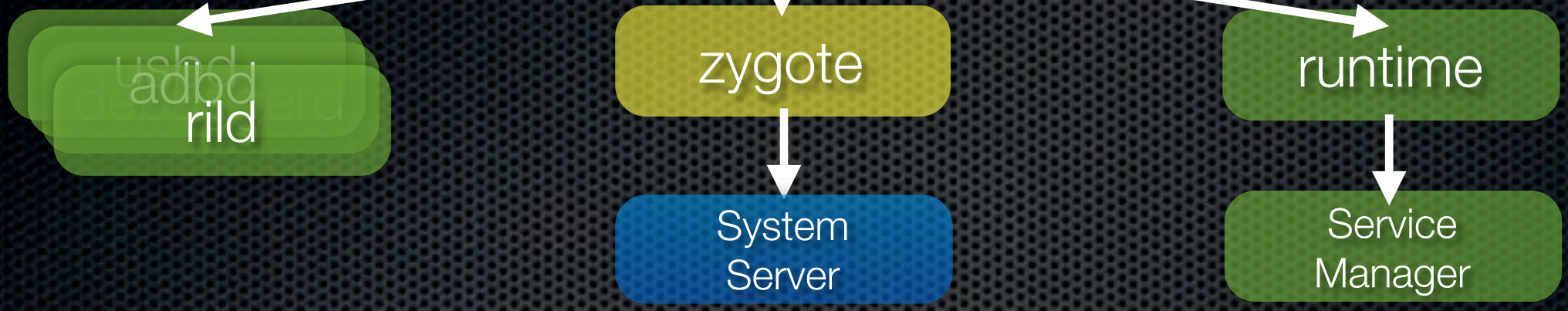
https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home

https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home

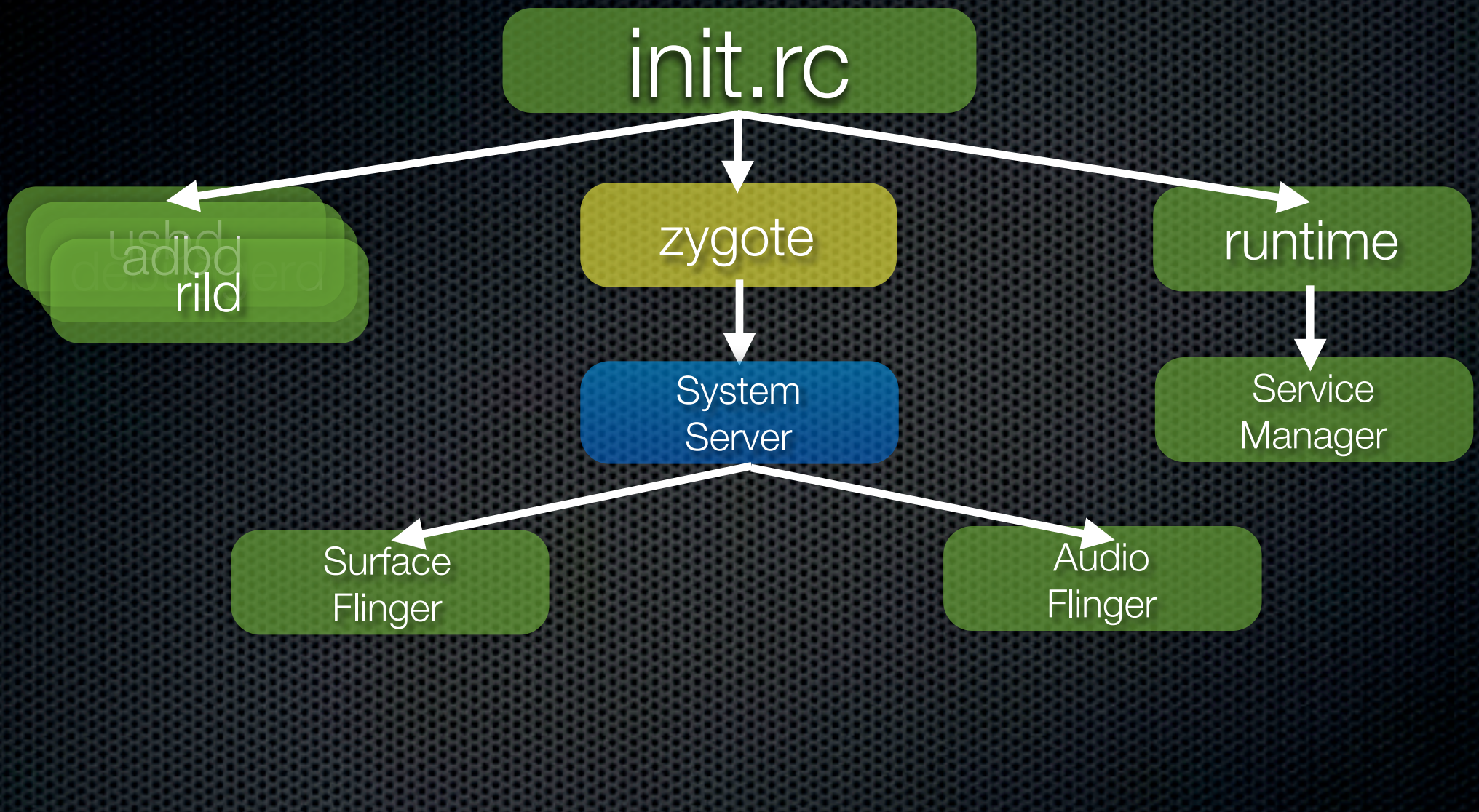
https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



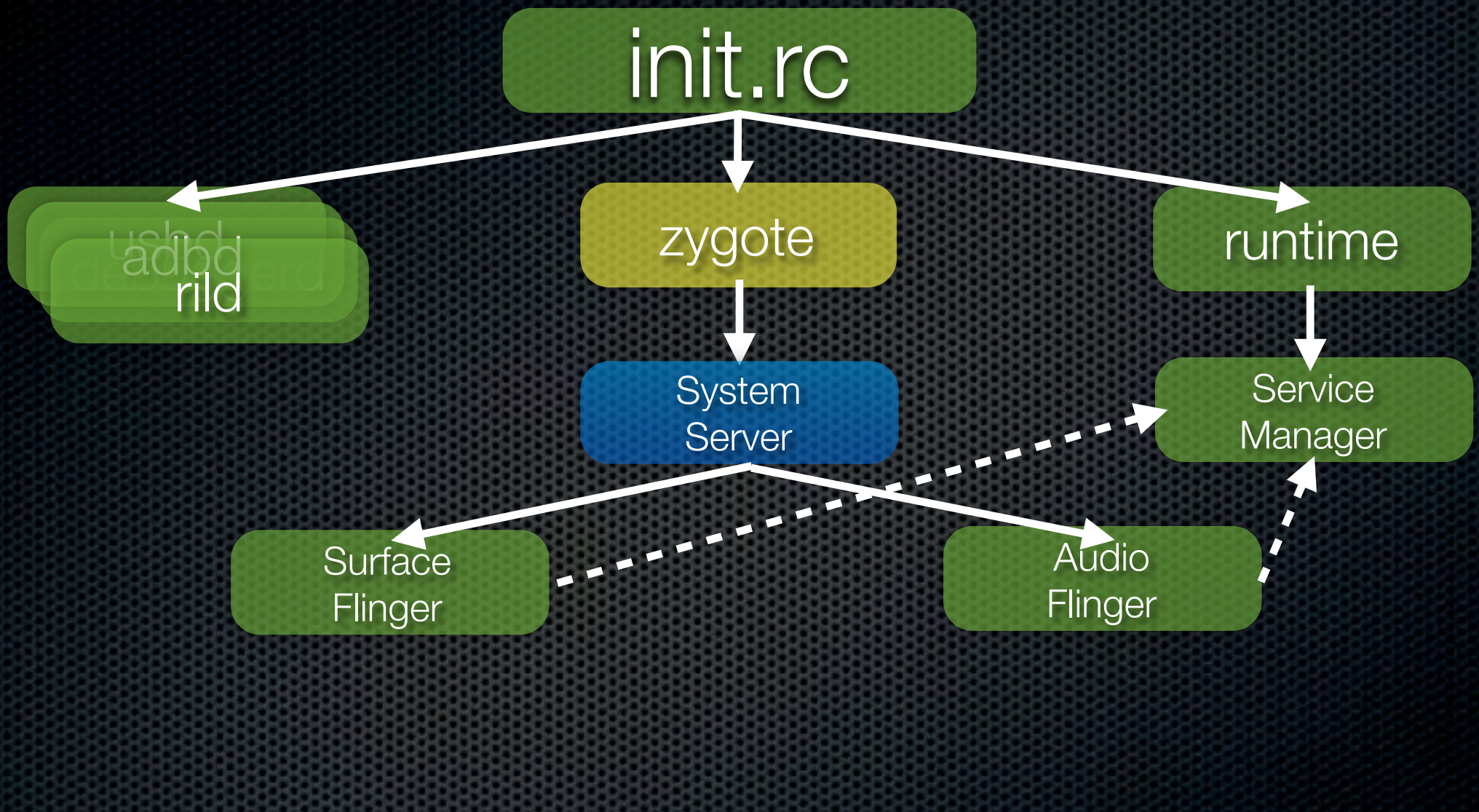
https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



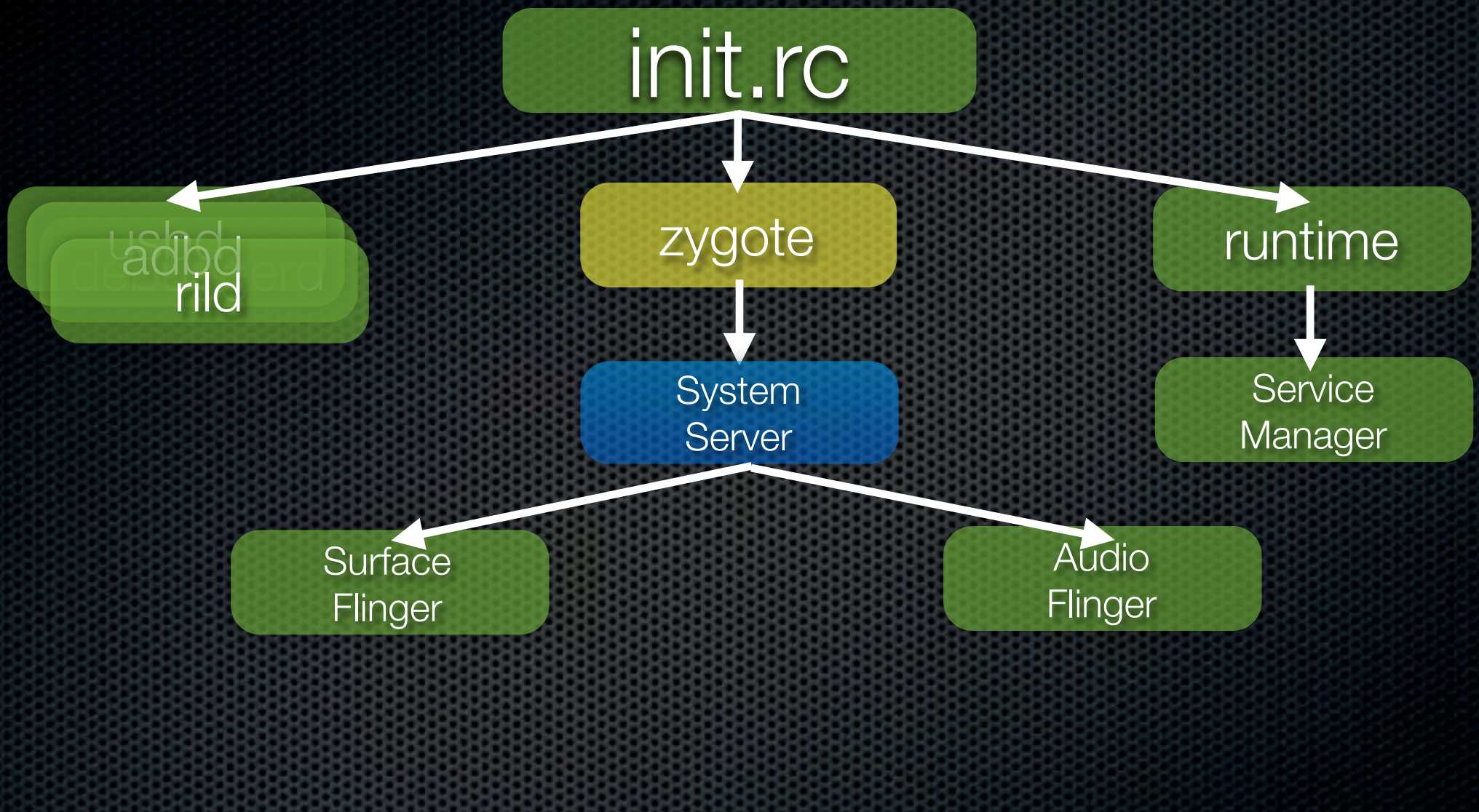
https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



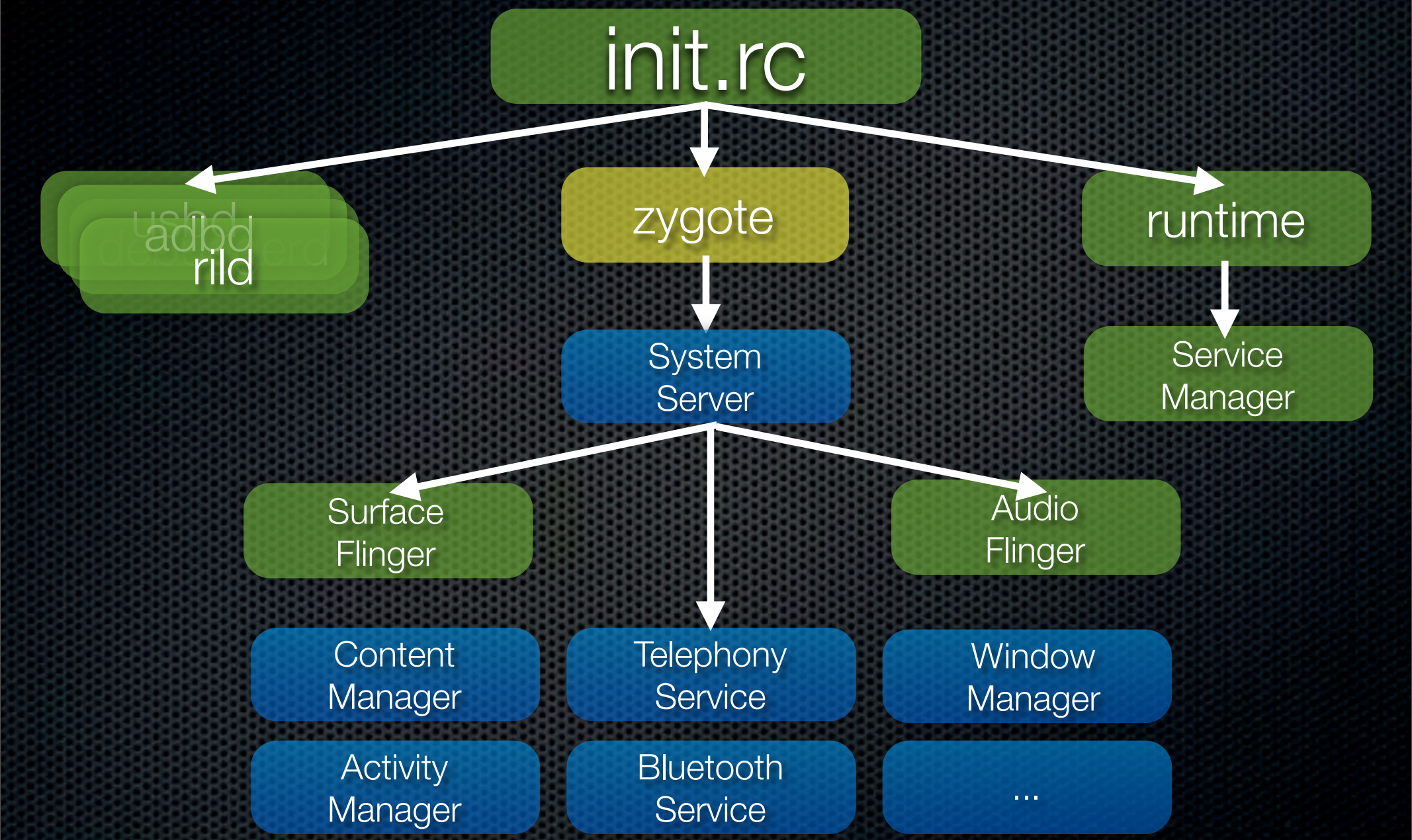
https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



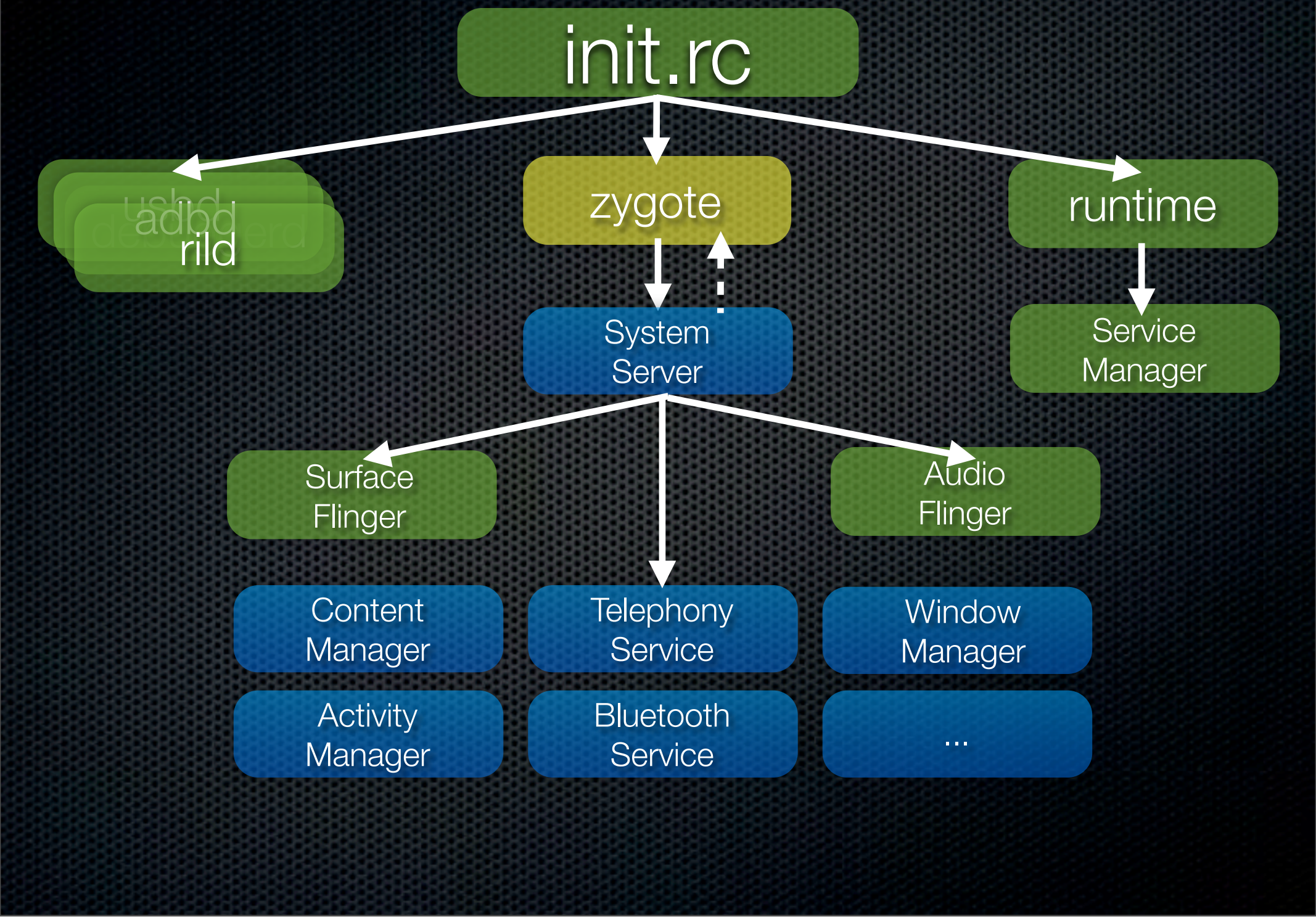
https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



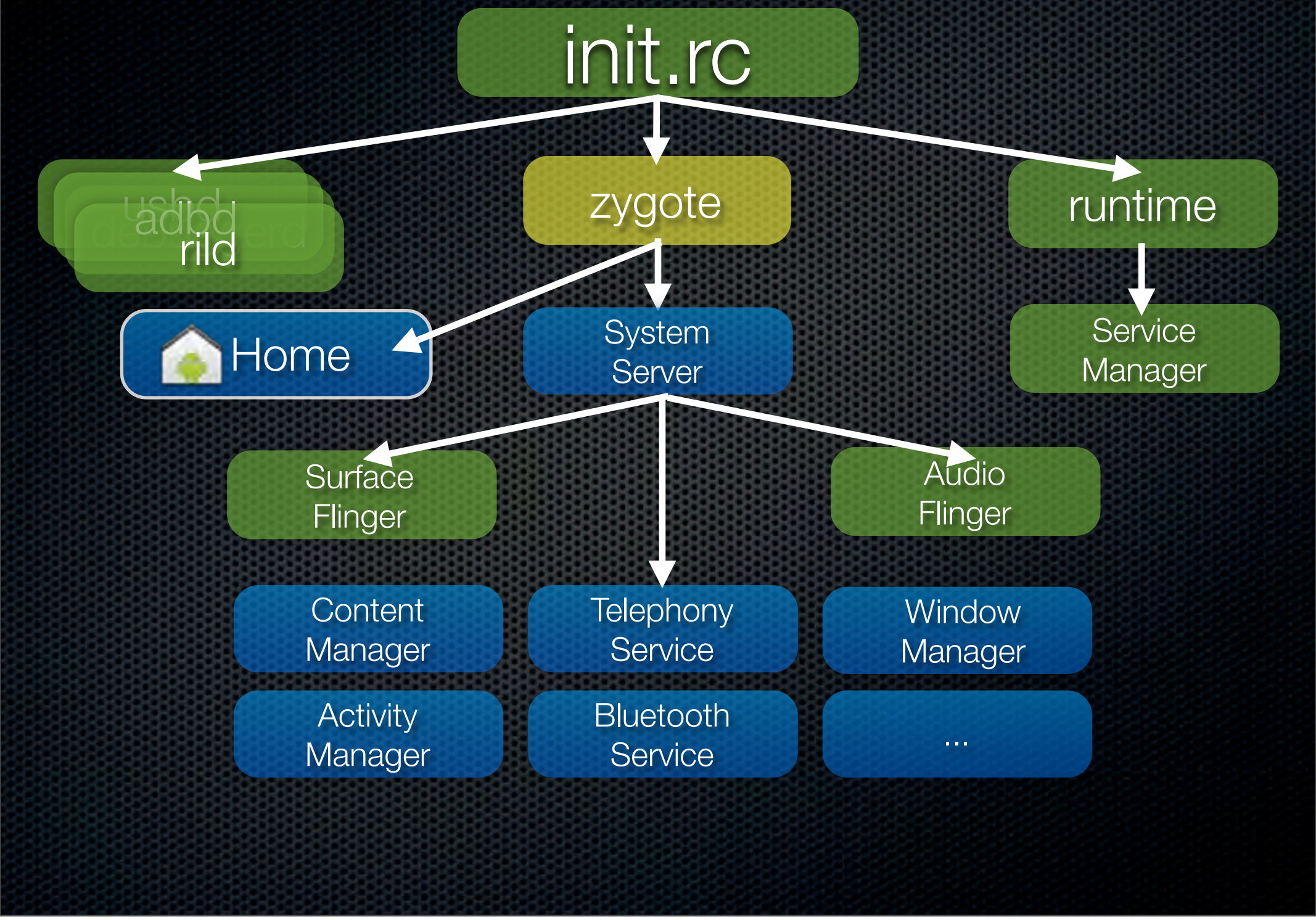
https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home



https://sites.google.com/site/io/anatomy--physiology-of-an-android
usbd - usb daemon
adb - android debug bridge
debuggerd - handles process dumps
rild - radio interface layer daemon
runtime - starts the Android service stack
runtime - sends signal to zygote to start System Server
SurfaceFlinger and AudioFlinger - started by System Server
Register back to ServiceManager
Other services started - Java proxies to telephony and bluetooth, etc.
Also register back to ServiceManager
Finally, another signal is sent to zygote and it starts Home

Demo

~5 minutes

Service List

```
> service list
0   sip: [android.net.sip.ISipService]
1   phone: [com.android.internal.telephony.ITelephony]
2   iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
3   simphonebook: [com.android.internal.telephony.IIccPhoneBook]
4   isms: [com.android.internal.telephony.ISms]
5   samplingprofiler: []
6 diskstats: []
```


Service listing shows some of the services that you could see in the previous diagram

init.rc

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote
    --start-system-service
    class main
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart surfaceflinger
    onrestart restart media
```


In init.rc, we can see zygote
We can see that its restart triggers surfaceflinger and audioflinger
Also, can see that it sets a wake lock

Bionic

Not glibc!

BSD License

Small

Harmony

Apache License

https://sites.google.com/site/io/anatomy--physiology-of-an-android

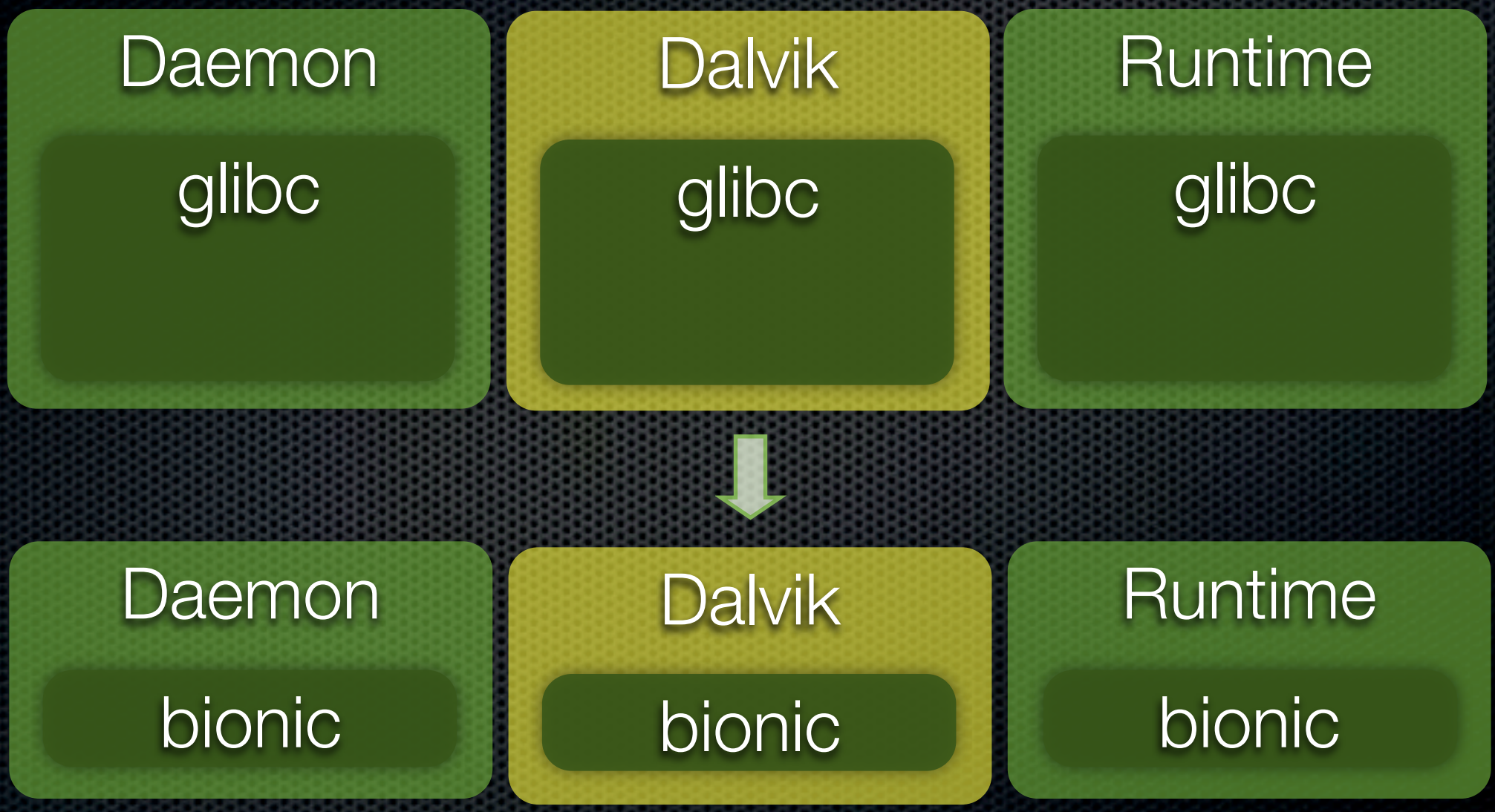
Want to avoid copyleft for benefit of hardware manufacturers

So, use bionic which is from BSD -- not fully glibc compatible -- no exception support - must use NDK

Harmony instead of standard Java libraries

Not Sun/Oracle certified -- Apache licensed

Bionic



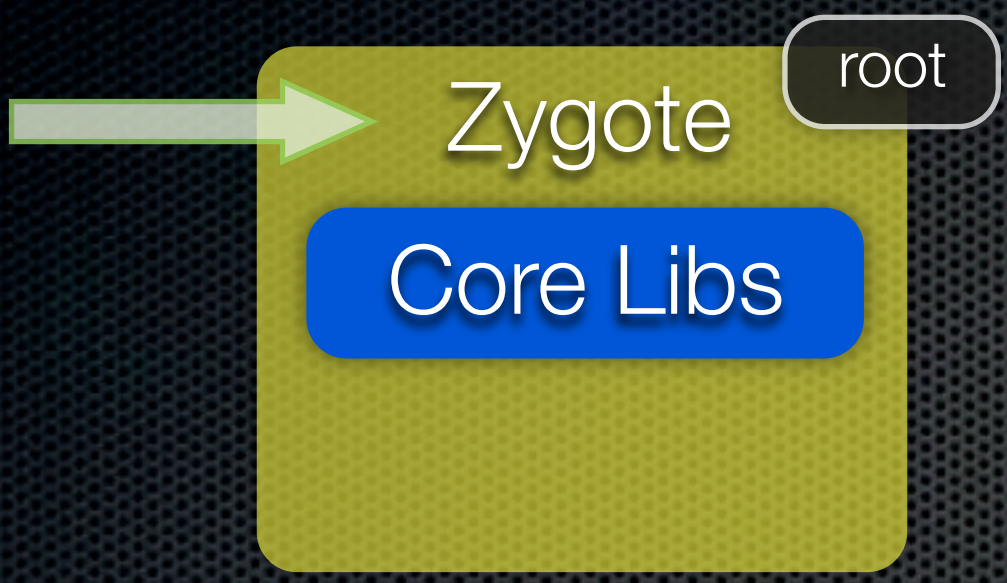
https://sites.google.com/site/io/anatomy--physiology-of-an-android
All processes listed earlier include libc, so using bionic makes everything trimmer
OS on Android -- uses 40MB
Other services -- 20MB

What's Zygote?

Zygote | Dalvik | ?

Three forms of Dalvik - normal Dalvik, Zygote, and one to be revealed later

Zygote

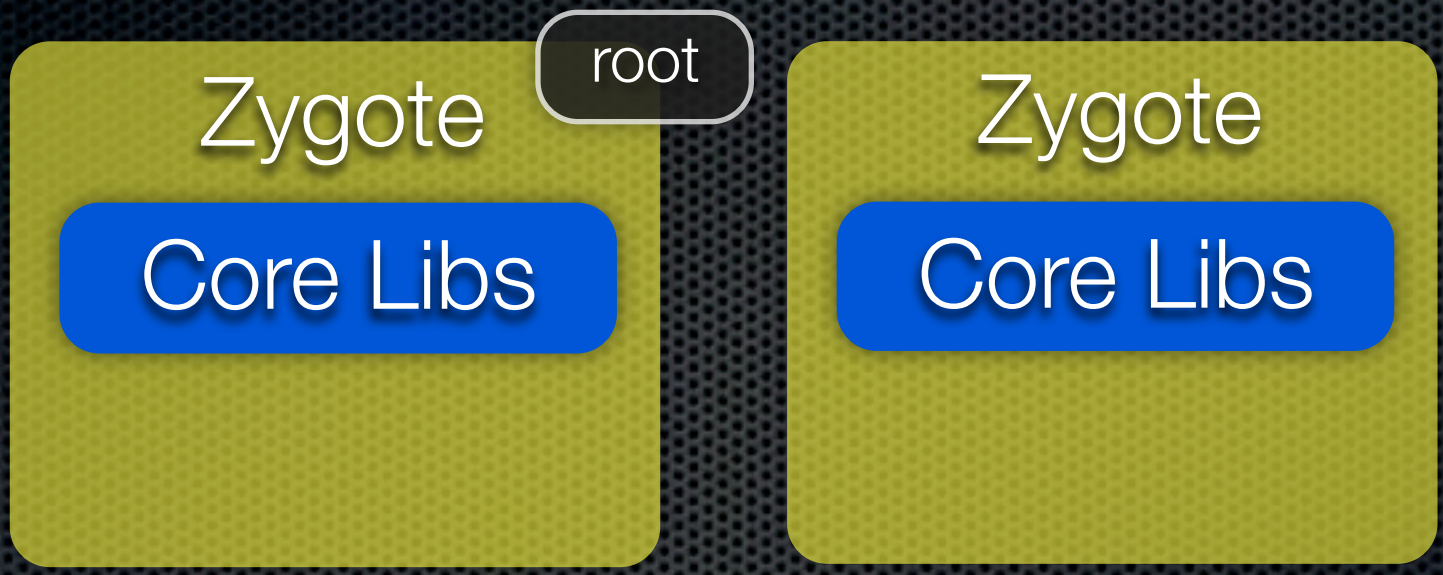


http://www.youtube.com/watch?v=ptjedOZEXPM
Nascent VM process from which other VM processes are spawned
Already has core libraries loaded in it
Runs as root

When an app is launched, zygote is forked
Fork core libraries are shared with zygote
App code is loaded into the forked VM
Ownership of the process is changed to a user generated for the app at install
Similar to having separate users from apache and mysql on a web server

The zygote means solves both the HotSpot problems: start-up performance and memory pressure

Zygote

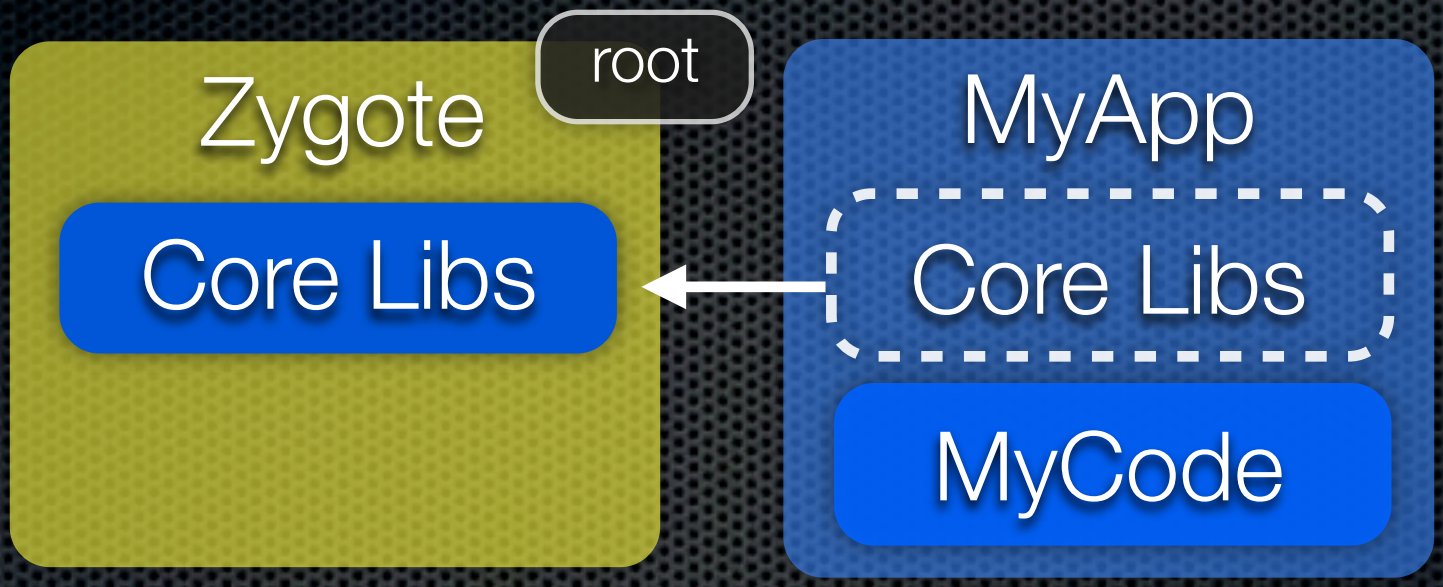


http://www.youtube.com/watch?v=ptjedOZEXPM
Nascent VM process from which other VM processes are spawned
Already has core libraries loaded in it
Runs as root

When an app is launched, zygote is forked
Fork core libraries are shared with zygote
App code is loaded into the forked VM
Ownership of the process is changed to a user generated for the app at install
Similar to having separate users from apache and mysql on a web server

The zygote means solves both the HotSpot problems: start-up performance and memory pressure

Zygote

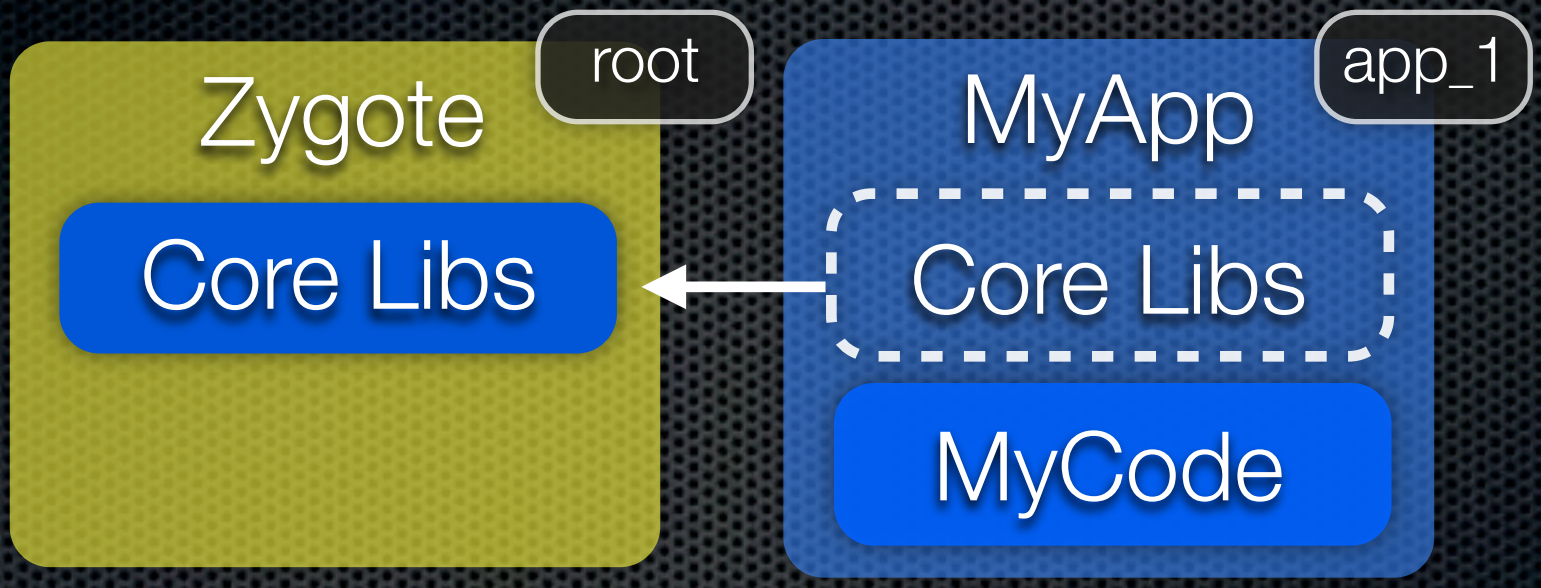


http://www.youtube.com/watch?v=ptjedOZEXPM
Nascent VM process from which other VM processes are spawned
Already has core libraries loaded in it
Runs as root

When an app is launched, zygote is forked
Fork core libraries are shared with zygote
App code is loaded into the forked VM
Ownership of the process is changed to a user generated for the app at install
Similar to having separate users from apache and mysql on a web server

The zygote means solves both the HotSpot problems: start-up performance and memory pressure

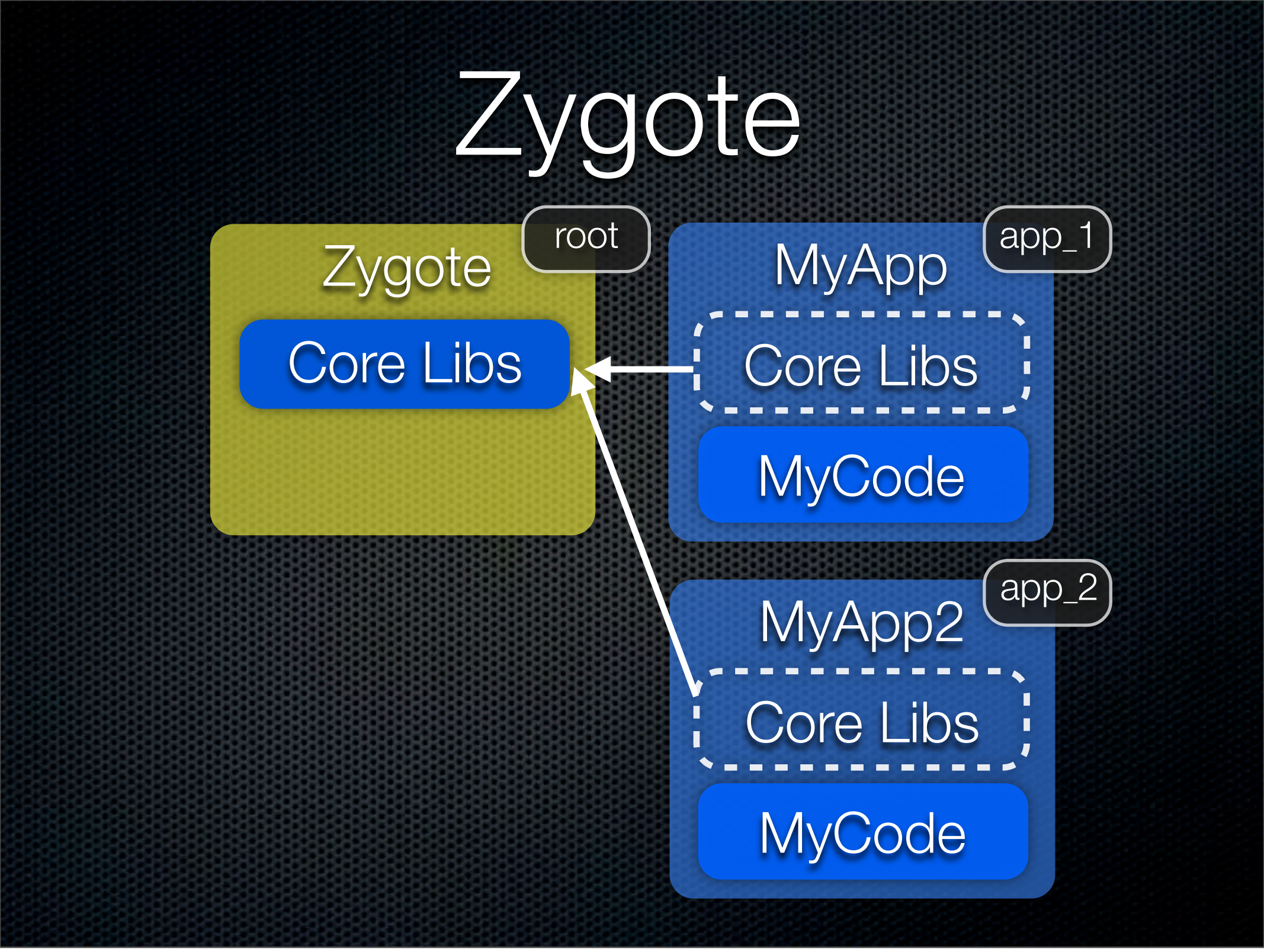
Zygote



http://www.youtube.com/watch?v=ptjedOZEXPM
Nascent VM process from which other VM processes are spawned
Already has core libraries loaded in it
Runs as root

When an app is launched, zygote is forked
Fork core libraries are shared with zygote
App code is loaded into the forked VM
Ownership of the process is changed to a user generated for the app at install
Similar to having separate users from apache and mysql on a web server

The zygote means solves both the HotSpot problems: start-up performance and memory pressure



http://www.youtube.com/watch?v=ptjedOZEXPM
Nascent VM process from which other VM processes are spawned
Already has core libraries loaded in it
Runs as root

When an app is launched, zygote is forked
Fork core libraries are shared with zygote
App code is loaded into the forked VM
Ownership of the process is changed to a user generated for the app at install
Similar to having separate users from apache and mysql on a web server

The zygote means solves both the HotSpot problems: start-up performance and memory pressure

http://www.youtube.com/watch?v=ptjedOZEXPM
Can split memory across 2-axes: Clean vs Dirty - Shared vs Private

Ideal is Clean - Shared
Core libraries loaded by zygote fall in this category
Can be reloaded by OS at will if need be - shared between processes

Clean - Private is good
This would be your application code -- not shared, but can be reloaded

Shared - Dirty is okay
Core libraries static variables might fall in this category, but split when process alters it

Private - Dirty is bad
Unfortunately, this is only thing you can create, so keep it to a minimum

Demo

~5 minutes

Processes

> ps

USER	PID	PPID	VSIZE	NAME
root	67	0	0	binder
system	82	820	272	/system/bin/servicemanager
root	83	4260	852	/system/bin/vold
root	84	4976	708	/system/bin/netd
root	85	684	252	/system/bin/debuggerd
system	86	20884	6900	/system/bin/surfaceflinger
root	87	419512	32228	zygote
app_20	13786	87	26304	com.google.android.apps.maps:NetworkLocationService
app_31	14558	87	25884	com.android.gallery3d
app_45	14571	87	25148	com.google.android.apps.books
app_55	14601	87	29560	com.twitter.android
app_16	16723	87	25384	com.google.android.music

In this process listing, you can see some of the daemons started at system start
Can see a number of Dalvik processes, each own by its app user and whose parent process is zygote

DB Ownership

```
USER      PID   PPID  VSIZE  NAME
app_50    1782  87    25384  com.google.android.deskclock
```

```
> ls /data/data/com.google.android.deskclock/databases
-rw-rw-r-- app_50  app_50     5857 2011-11-06 17:29 alarms.db
-rw-rw-r-- app_50  app_50   153432 2011-11-06 17:23 alarms.db-journal
```


A data directory is created for each app...
SQLite databases created by each app are owned by the same user id used when the app is running

Dalvik Does Not Do Security

Warning: security managers do **not** provide a secure environment for executing untrusted code. Untrusted code cannot be safely isolated within the Dalvik VM.

Because the system handles app isolation, Dalvik does not.
http://developer.android.com/guide/topics/security/security.html
http://developer.android.com/reference/java/lang/SecurityManager.html

Isolation is Good...

...but Apps need to Share.

Ashmem & Binder

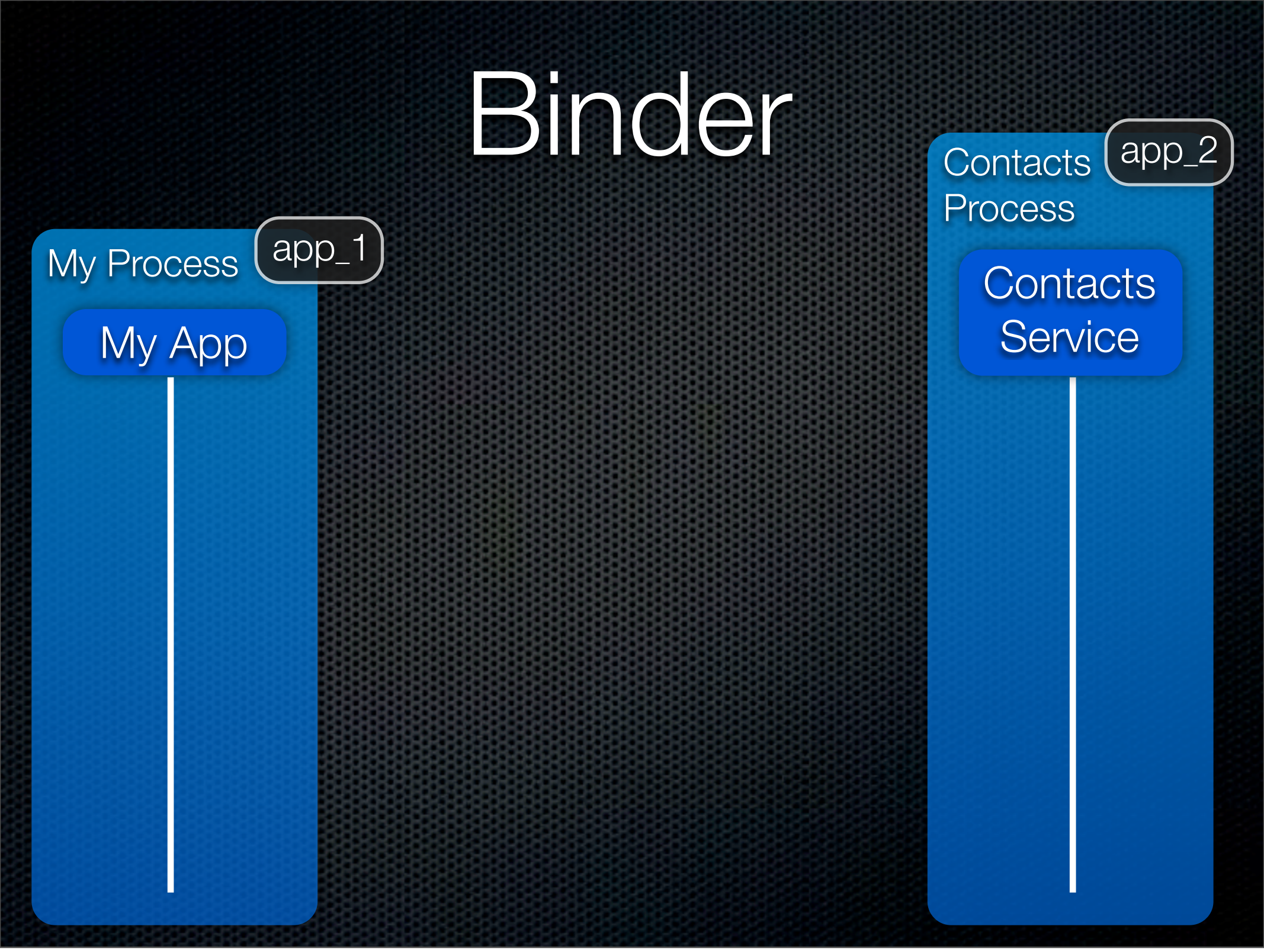
Android Shared Memory

Reference Counts

Binder - derived from OpenBinder

From BeOS - OO IPC

http://www.lindusembedded.com/blog/2010/12/07/android-linux-kernel-additions/
http://www.osnews.com/story/13674/

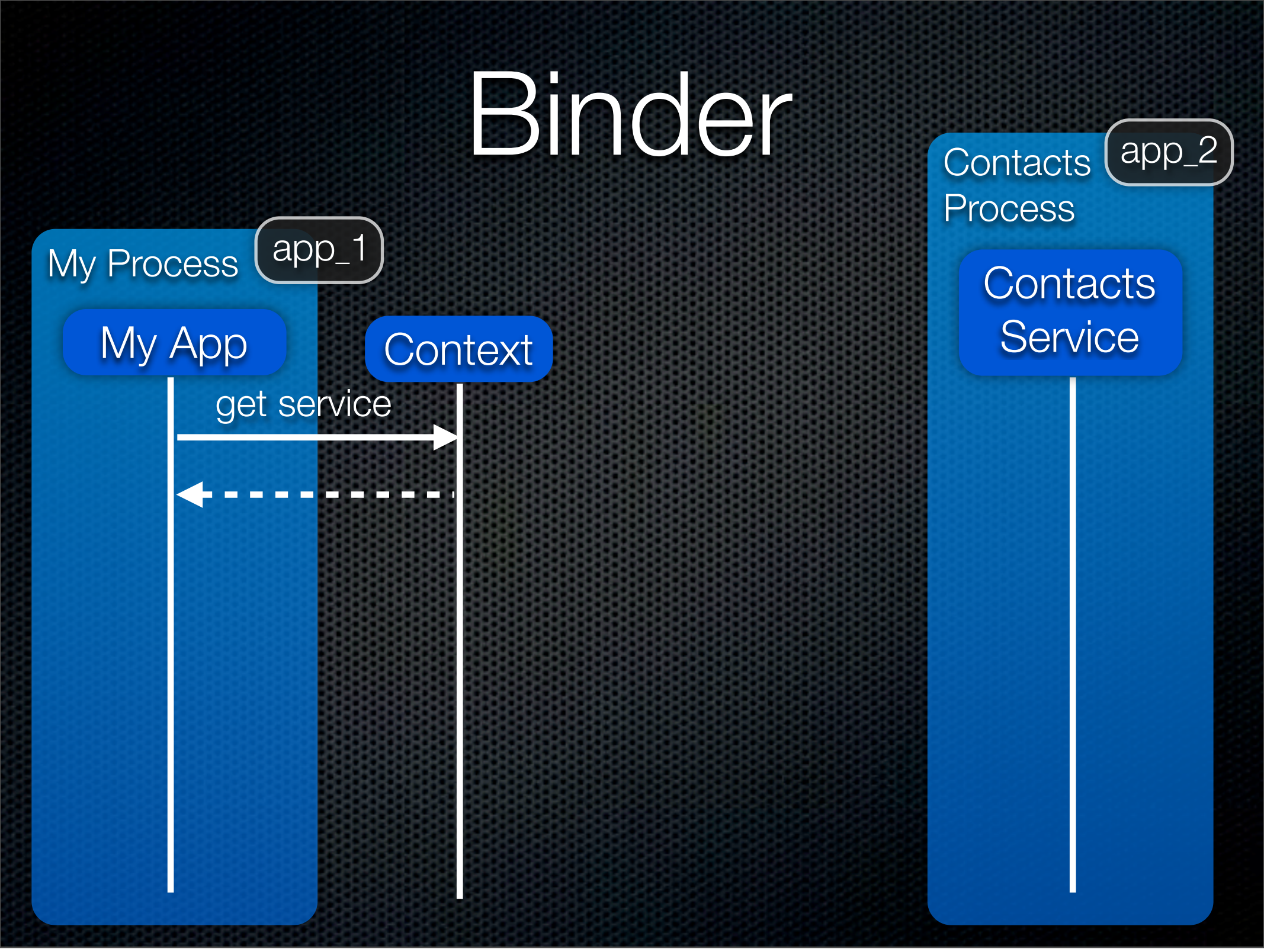


https://sites.google.com/site/io/anatomy--physiology-of-an-android
How do apps talk to each other?

Consider “My App” trying to use the “Contacts Service” -- each are running as separate processes
“My App” uses the “Context” object to locate a service and it gets a reference

However, the object it is given is a proxy managed via Binder
When “My App” tries to invoke “query”, Binder intervenes and marshals the request over shared memory

However, the request does not go to the main application thread because we don’t that app or other apps to start
So, the Binder system creates a pool of threads for handling service requests
The “query” method is executed by a service thread, the result is marshalled over shared memory and ultimately back to “My App”

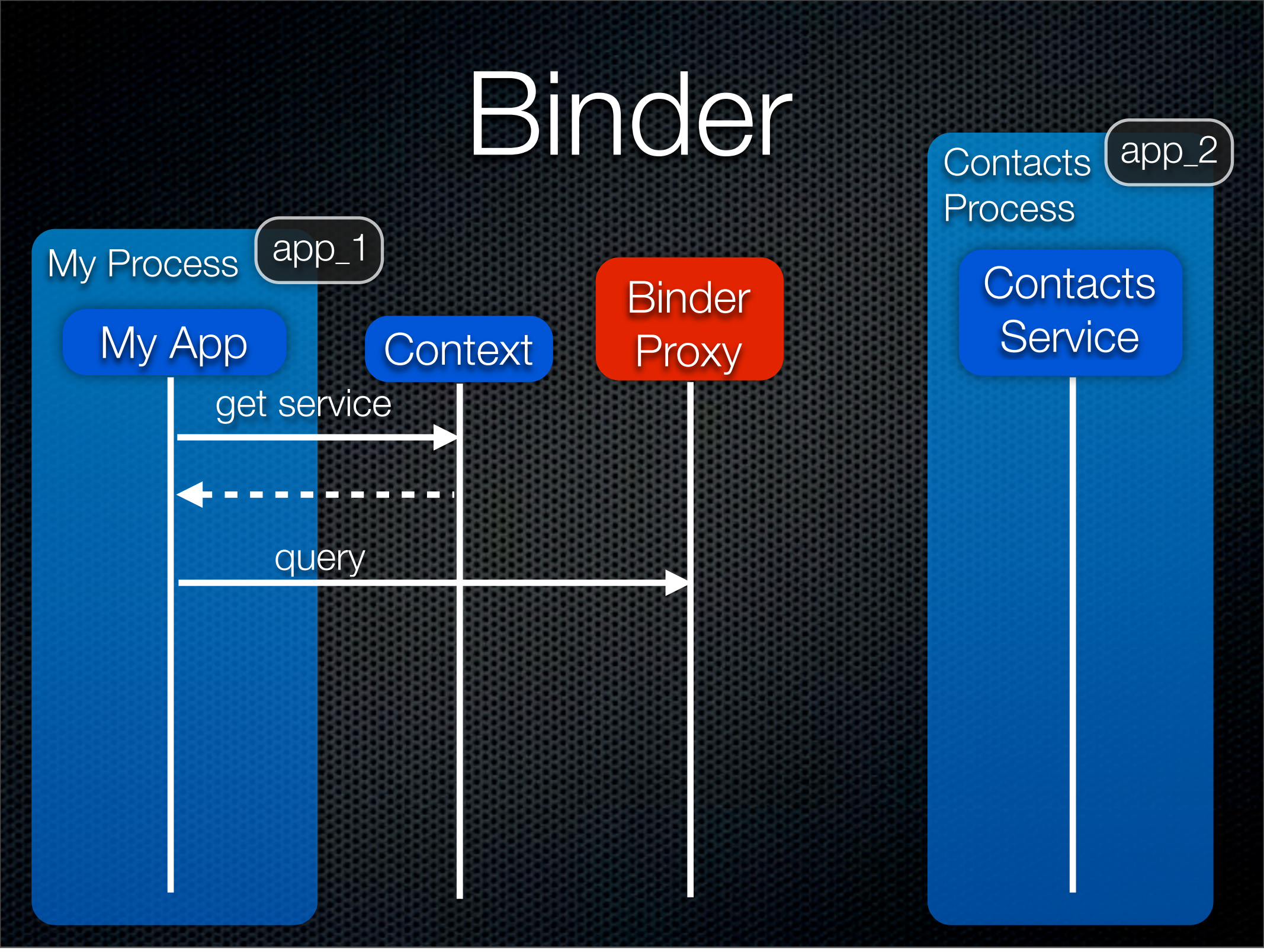


https://sites.google.com/site/io/anatomy--physiology-of-an-android
How do apps talk to each other?

Consider “My App” trying to use the “Contacts Service” -- each are running as separate processes
“My App” uses the “Context” object to locate a service and it gets a reference

However, the object it is given is a proxy managed via Binder
When “My App” tries to invoke “query”, Binder intervenes and marshals the request over shared memory

However, the request does not go to the main application thread because we don’t that app or other apps to start
So, the Binder system creates a pool of threads for handling service requests
The “query” method is executed by a service thread, the result is marshalled over shared memory and ultimately back to “My App”

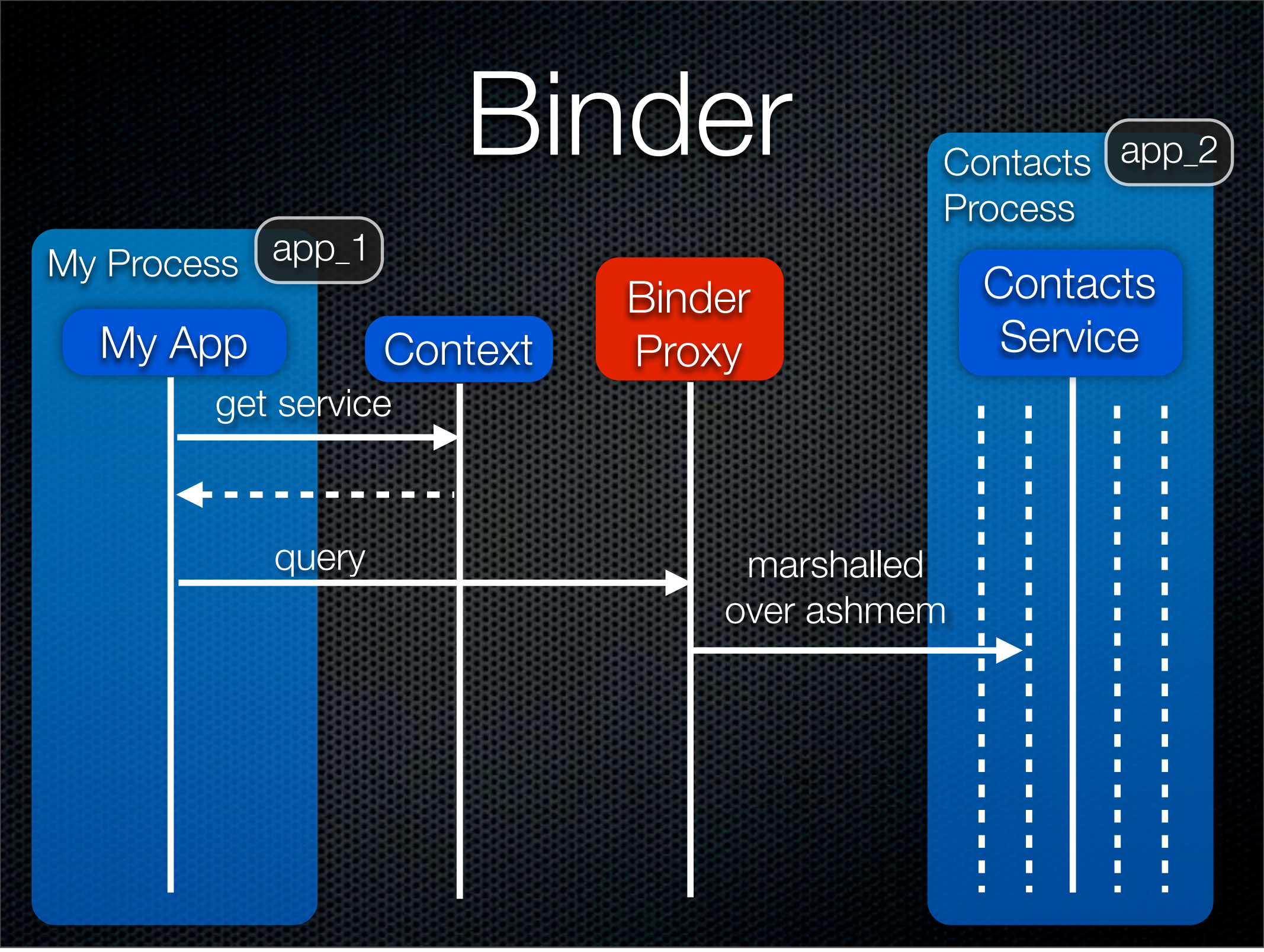


https://sites.google.com/site/io/anatomy--physiology-of-an-android
How do apps talk to each other?

Consider “My App” trying to use the “Contacts Service” -- each are running as separate processes
“My App” uses the “Context” object to locate a service and it gets a reference

However, the object it is given is a proxy managed via Binder
When “My App” tries to invoke “query”, Binder intervenes and marshals the request over shared memory

However, the request does not go to the main application thread because we don’t that app or other apps to start
So, the Binder system creates a pool of threads for handling service requests
The “query” method is executed by a service thread, the result is marshalled over shared memory and ultimately back to “My App”

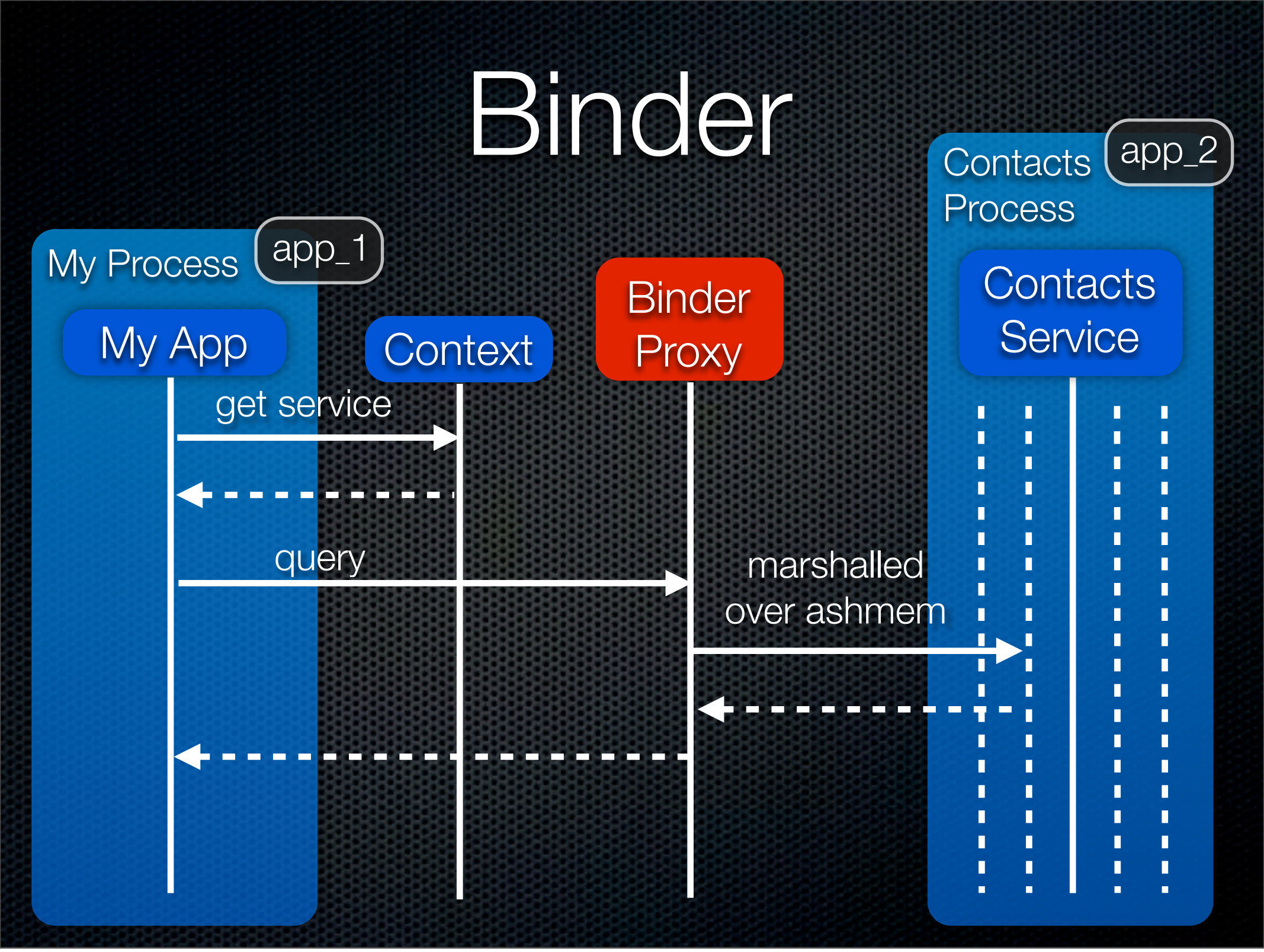


https://sites.google.com/site/io/anatomy--physiology-of-an-android
How do apps talk to each other?

Consider “My App” trying to use the “Contacts Service” -- each are running as separate processes
“My App” uses the “Context” object to locate a service and it gets a reference

However, the object it is given is a proxy managed via Binder
When “My App” tries to invoke “query”, Binder intervenes and marshals the request over shared memory

However, the request does not go to the main application thread because we don’t that app or other apps to start
So, the Binder system creates a pool of threads for handling service requests
The “query” method is executed by a service thread, the result is marshalled over shared memory and ultimately back to “My App”



https://sites.google.com/site/io/anatomy--physiology-of-an-android
How do apps talk to each other?

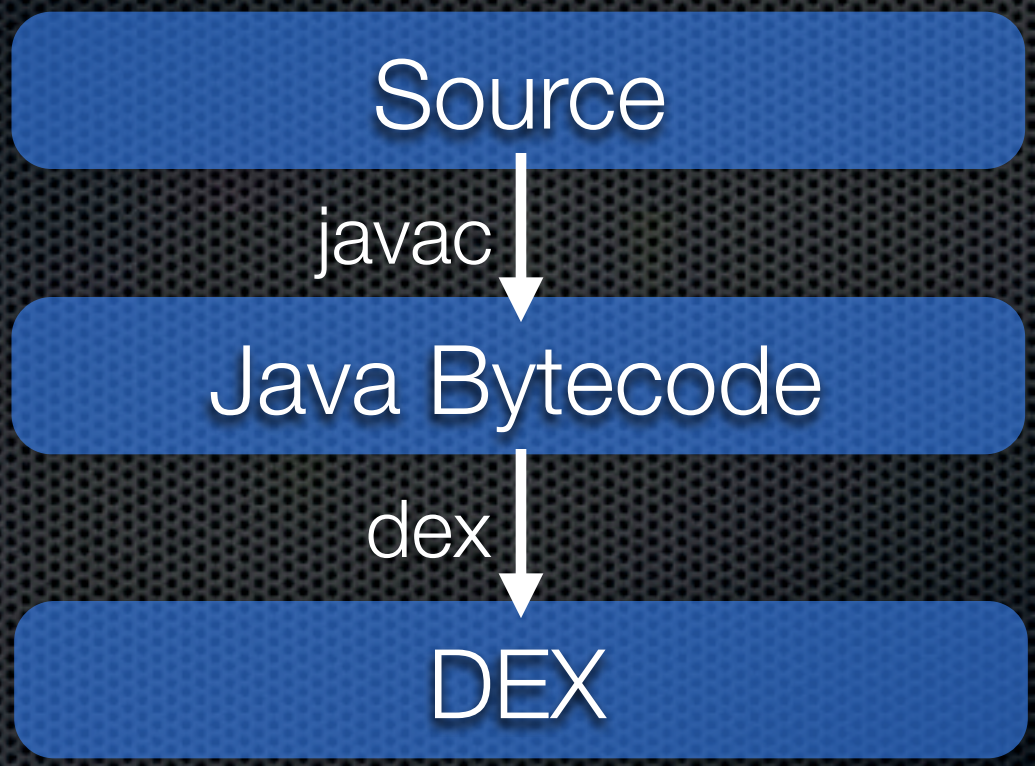
Consider “My App” trying to use the “Contacts Service” -- each are running as separate processes
“My App” uses the “Context” object to locate a service and it gets a reference

However, the object it is given is a proxy managed via Binder
When “My App” tries to invoke “query”, Binder intervenes and marshals the request over shared memory

However, the request does not go to the main application thread because we don’t that app or other apps to start
So, the Binder system creates a pool of threads for handling service requests
The “query” method is executed by a service thread, the result is marshalled over shared memory and ultimately back to “My App”

Dalvik

Different Bytecode



http://www.youtube.com/watch?v=ptjedOZEXPM

Android does an extra translation step to a DEX (which is wrapped in an APK)

This is done by the DEX tool (which is actually written in Java)

File Format

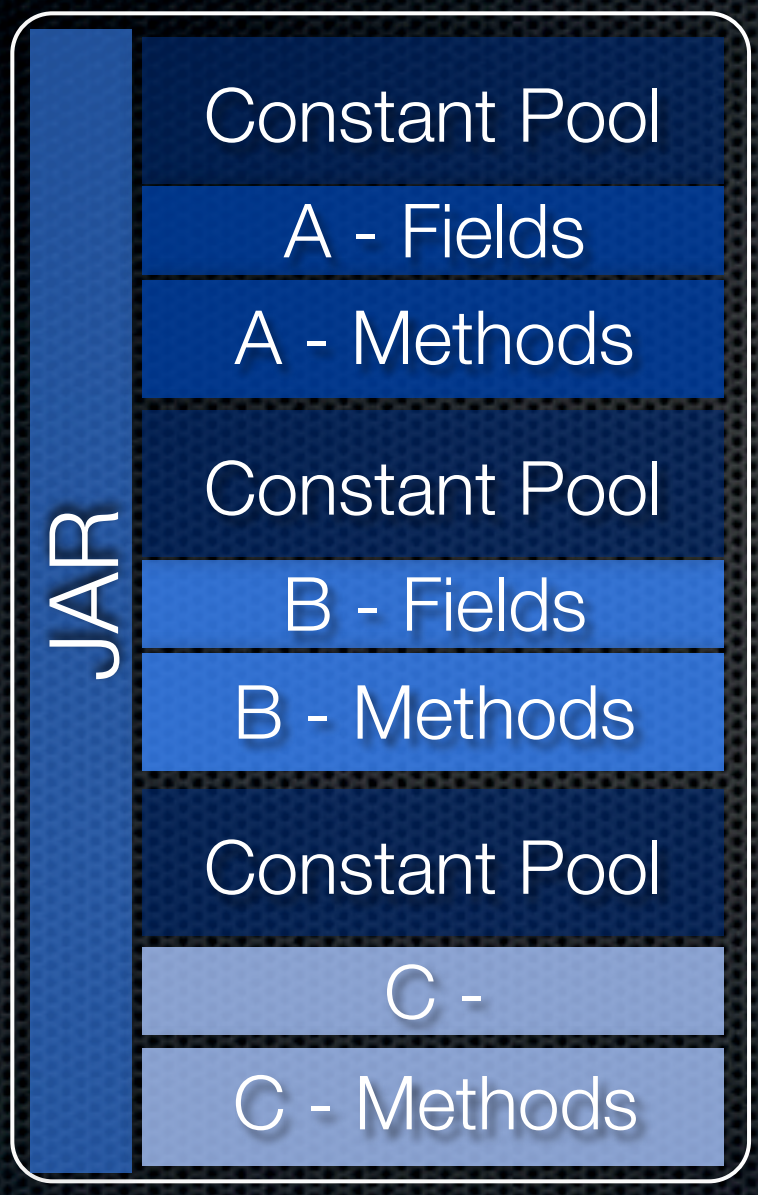
http://www.youtube.com/watch?v=ptjedOZEXPM
A Java Class file consists of a Constant Pool, Fields, and Methods
Constant Pool is a heterogenous mix of numbers, Strings, class references, method references, etc.
We might pack a few of these into a JAR file

The problem with this is that common elements (String for example) will be repeated in all the constant pools.

So, Dalvik takes a different approach. It merges all the constant pools together into a single constant pool.
Then, it mixes all the fields. Then, all the methods.

This way there's only a single reference to commonly used classes methods, etc. This saves space.
Also, the constant pool is not heterogenous mix. It is segmented into strings, then

File Format



http://www.youtube.com/watch?v=ptjedOZEXPM

A Java Class file consists of a Constant Pool, Fields, and Methods

Constant Pool is a heterogenous mix of numbers, Strings, class references, method references, etc.

We might pack a few of these into a JAR file

The problem with this is that common elements (String for example) will be repeated in all the constant pools.

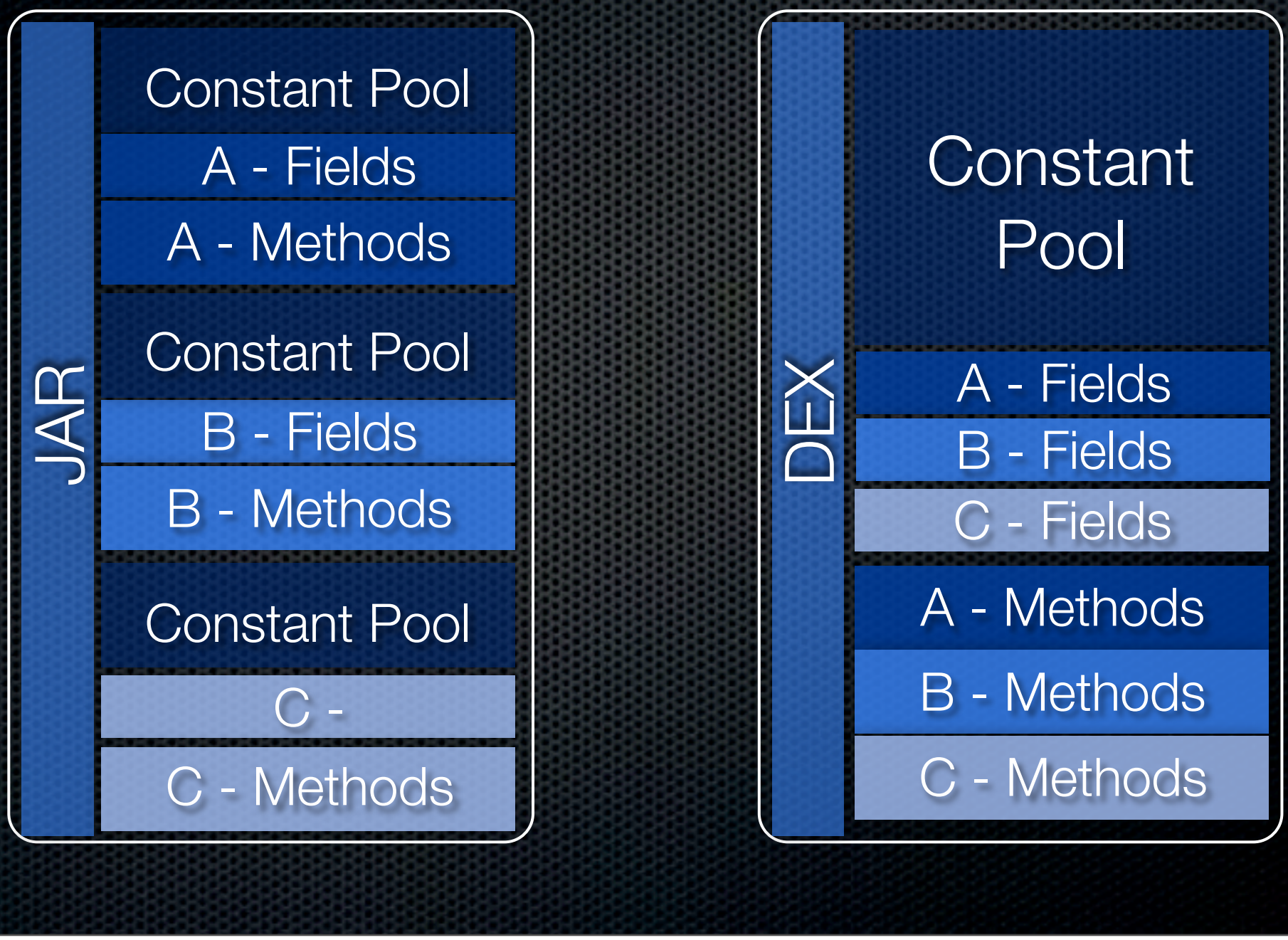
So, Dalvik takes a different approach. It merges all the constant pools together into a single constant pool.

Then, it mixes all the fields. Then, all the methods.

This way there's only a single reference to commonly used classes methods, etc. This saves space.

Also, the constant pool is not heterogenous mix. It is segmented into strings, then

File Format



http://www.youtube.com/watch?v=ptjedOZEXPM

A Java Class file consists of a Constant Pool, Fields, and Methods

Constant Pool is a heterogenous mix of numbers, Strings, class references, method references, etc.

We might pack a few of these into a JAR file

The problem with this is that common elements (String for example) will be repeated in all the constant pools.

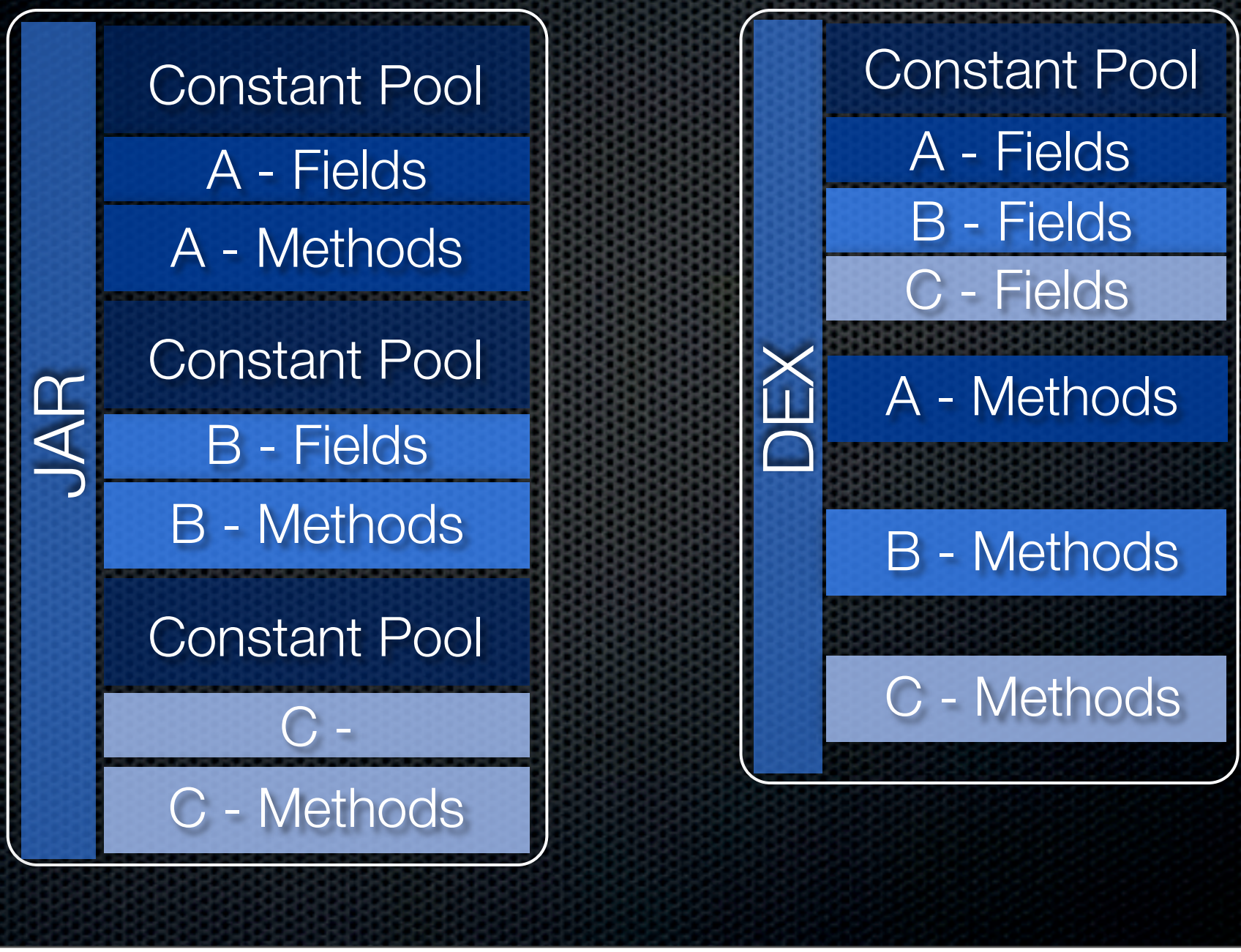
So, Dalvik takes a different approach. It merges all the constant pools together into a single constant pool.

Then, it mixes all the fields. Then, all the methods.

This way there's only a single reference to commonly used classes methods, etc. This saves space.

Also, the constant pool is not heterogenous mix. It is segmented into strings, then

File Size



http://www.youtube.com/watch?v=ptjedOZEXPM

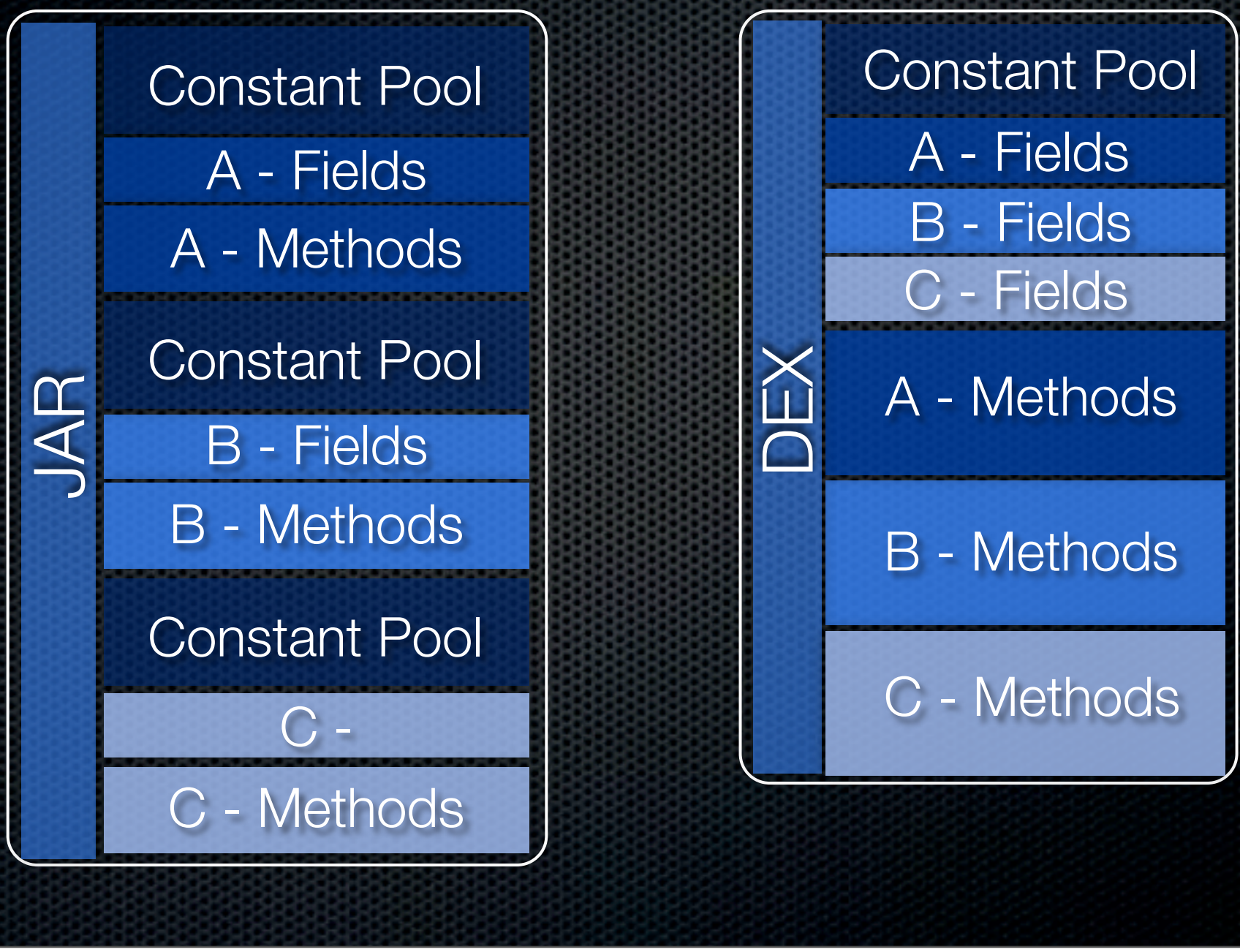
What's the next effect in terms of size?

Well, DEX save a fair amount of space by having a single constant pool, but, as we'll see, Dalvik byte code is a little bigger.

However, JARs are basically ZIPs, so they can be compressed. And, of course, those repetitions in the constant pools compress well and so does bytecode -- so in the end, an uncompressed DEX is about the same size as a compressed JAR.

However, the DEX is much easier for the interpreter to work with -- the interpreter can easily memory map the file and randomly access it -- rather than streaming the file.

File Size



http://www.youtube.com/watch?v=ptjedOZEXPM

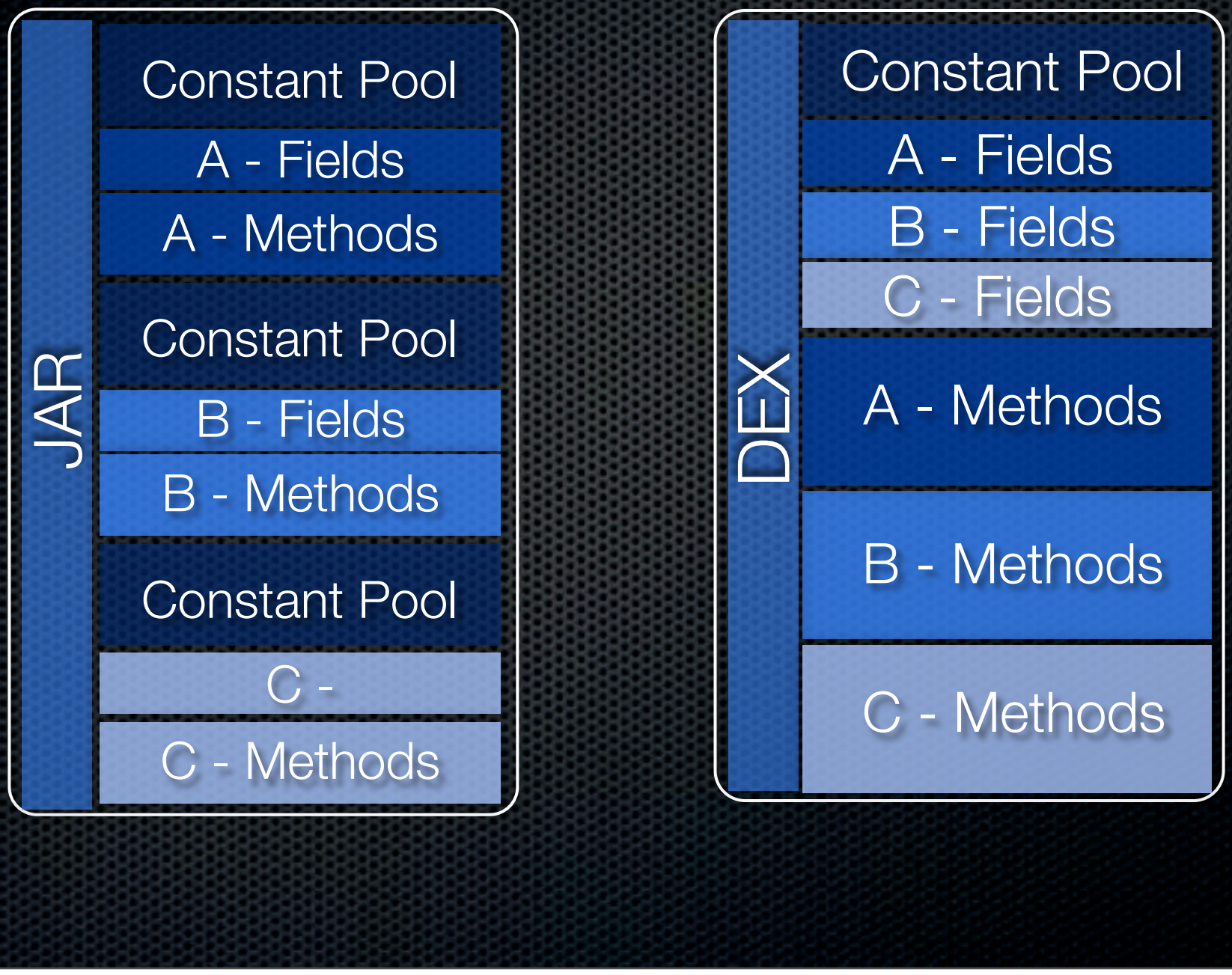
What's the next effect in terms of size?

Well, DEX save a fair amount of space by having a single constant pool, but, as we'll see, Dalvik byte code is a little bigger.

However, JARs are basically ZIPs, so they can be compressed. And, of course, those repetitions in the constant pools compress well and so does bytecode -- so in the end, an uncompressed DEX is about the same size as a compressed JAR.

However, the DEX is much easier for the interpreter to work with -- the interpreter can easily memory map the file and randomly access it -- rather than streaming the file.

File Size



http://www.youtube.com/watch?v=ptjedOZEXPM

What's the next effect in terms of size?

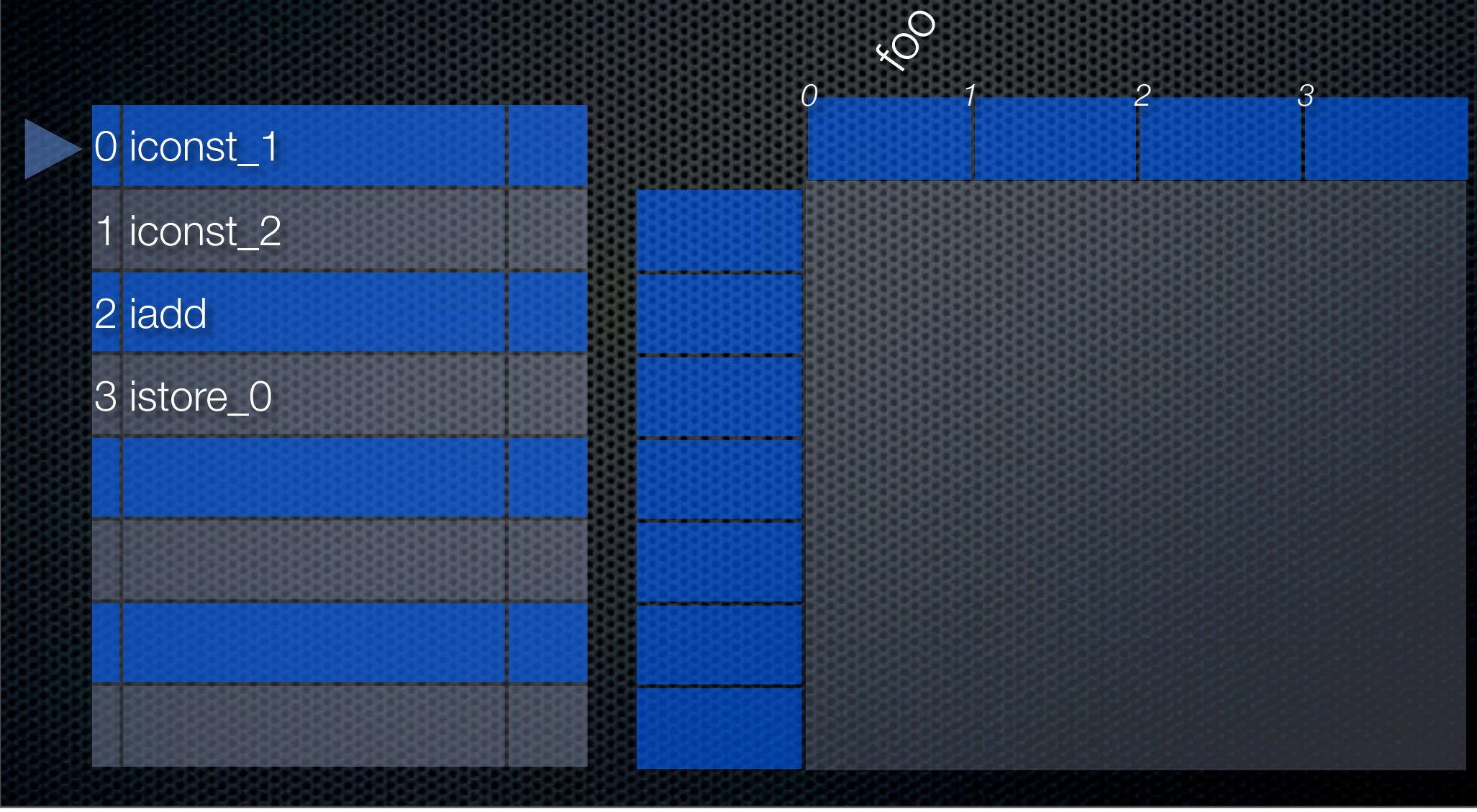
Well, DEX save a fair amount of space by having a single constant pool, but, as we'll see, Dalvik byte code is a little bigger.

However, JARs are basically ZIPs, so they can be compressed. And, of course, those repetitions in the constant pools compress well and so does bytecode -- so in the end, an uncompressed DEX is about the same size as a compressed JAR.

However, the DEX is much easier for the interpreter to work with -- the interpreter can easily memory map the file and randomly access it -- rather than streaming the file.

Java is Stack Based

int foo = 1+2;



Wednesday, November 9, 11

As you are probably aware, standard JVMs are stack-based.
So, consider the code “int foo = 1 + 2”, this translates to the bytecode shown on the left.
The gray field represents the heap, the column to its right is the stack, and the row above are the local variable slots.
In regular bytecode, we’d run...
iconst_1 to push 1 onto the stack
iconst_2 to push 2 onto the stack
iadd to sum the 1 and 2 on the stack and store the result 3
istore_0 to store the value in the stack into local variable slot 0 -- namely “foo”

Java is Stack Based

int foo = 1+2;

Wednesday, November 9, 11

As you are probably aware, standard JVMs are stack-based.
So, consider the code “int foo = 1 + 2”, this translates to the bytecode shown on the left.
The gray field represents the heap, the column to its right is the stack, and the row above are the local variable slots.
In regular bytecode, we’d run...
iconst_1 to push 1 onto the stack
iconst_2 to push 2 onto the stack
iadd to sum the 1 and 2 on the stack and store the result 3
istore_0 to store the value in the stack into local variable slot 0 -- namely “foo”

Java is Stack Based

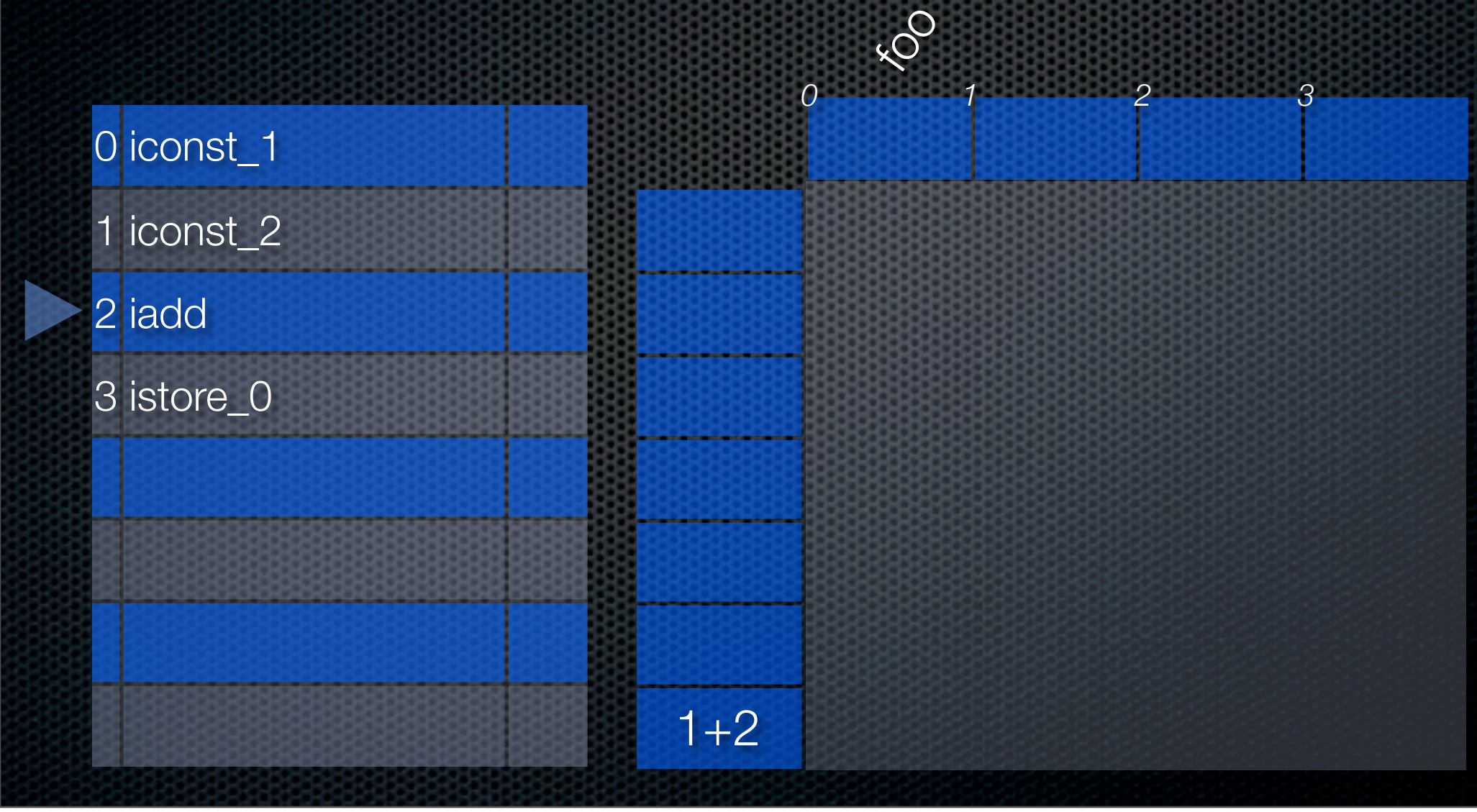
int foo = 1+2;

Wednesday, November 9, 11

As you are probably aware, standard JVMs are stack-based.
So, consider the code “int foo = 1 + 2”, this translates to the bytecode shown on the left.
The gray field represents the heap, the column to its right is the stack, and the row above are the local variable slots.
In regular bytecode, we’d run...
iconst_1 to push 1 onto the stack
iconst_2 to push 2 onto the stack
iadd to sum the 1 and 2 on the stack and store the result 3
istore_0 to store the value in the stack into local variable slot 0 -- namely “foo”

Java is Stack Based

int foo = 1+2;



Wednesday, November 9, 11

As you are probably aware, standard JVMs are stack-based.
So, consider the code “int foo = 1 + 2”, this translates to the bytecode shown on the left.
The gray field represents the heap, the column to its right is the stack, and the row above are the local variable slots.
In regular bytecode, we’d run...
iconst_1 to push 1 onto the stack
iconst_2 to push 2 onto the stack
iadd to sum the 1 and 2 on the stack and store the result 3
istore_0 to store the value in the stack into local variable slot 0 -- namely “foo”

Java is Stack Based

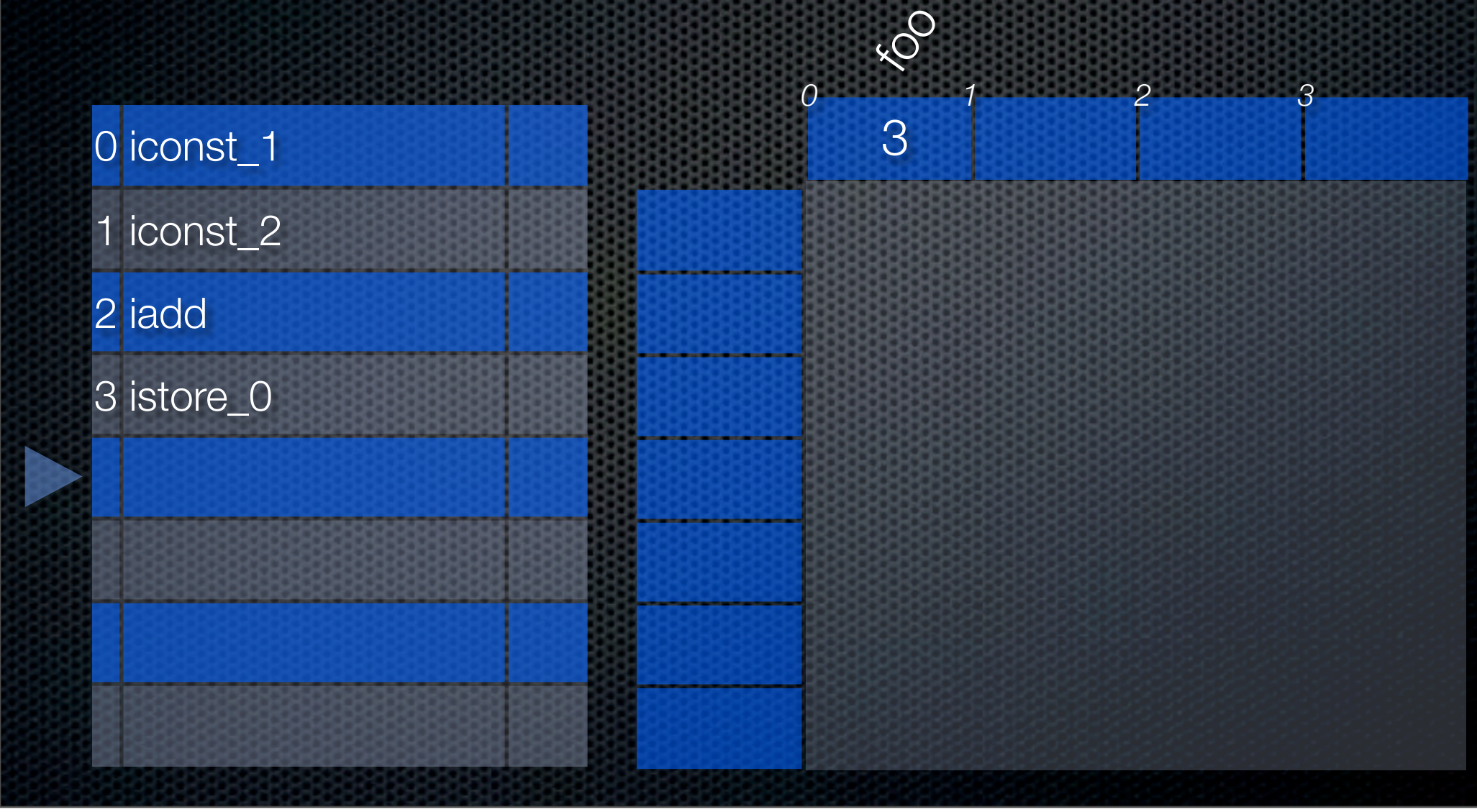
int foo = 1+2;

Wednesday, November 9, 11

As you are probably aware, standard JVMs are stack-based.
So, consider the code “int foo = 1 + 2”, this translates to the bytecode shown on the left.
The gray field represents the heap, the column to its right is the stack, and the row above are the local variable slots.
In regular bytecode, we’d run...
iconst_1 to push 1 onto the stack
iconst_2 to push 2 onto the stack
iadd to sum the 1 and 2 on the stack and store the result 3
istore_0 to store the value in the stack into local variable slot 0 -- namely “foo”

Java is Stack Based

int foo = 1+2;



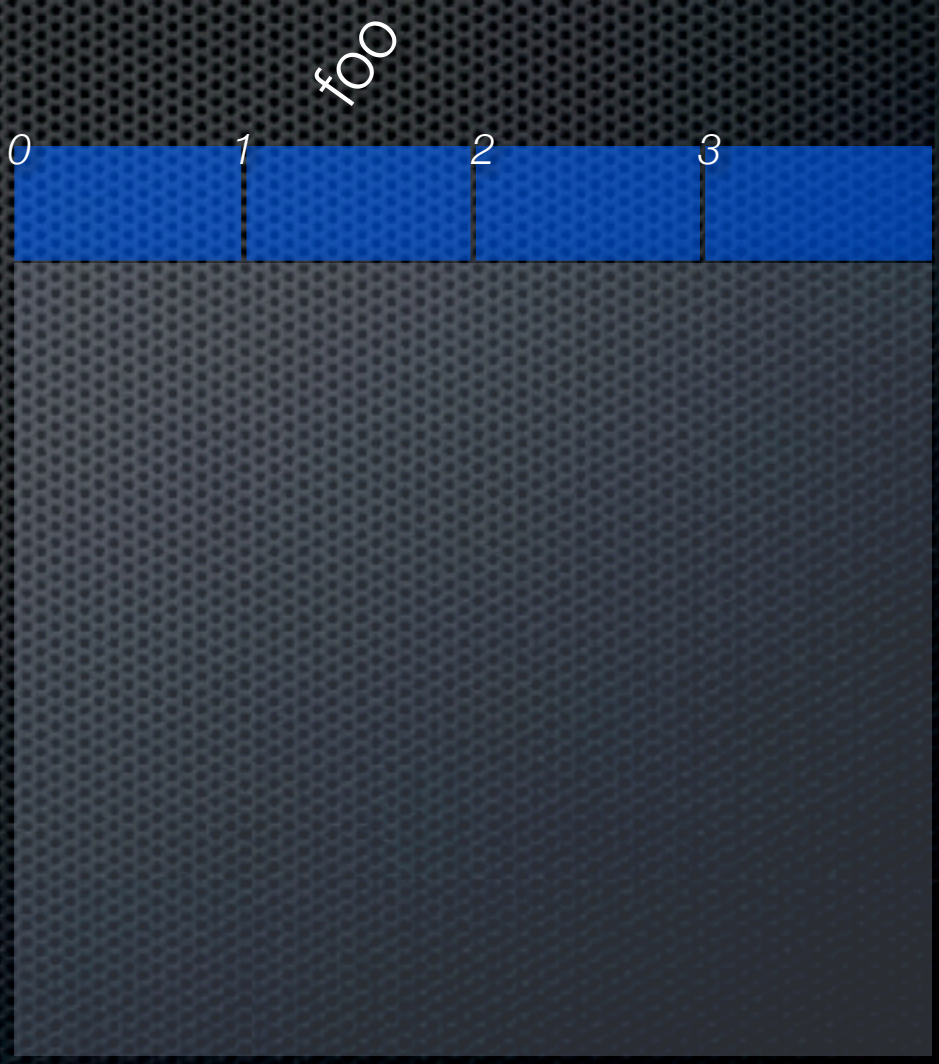
Wednesday, November 9, 11

As you are probably aware, standard JVMs are stack-based.
So, consider the code “int foo = 1 + 2”, this translates to the bytecode shown on the left.
The gray field represents the heap, the column to its right is the stack, and the row above are the local variable slots.
In regular bytecode, we’d run...
iconst_1 to push 1 onto the stack
iconst_2 to push 2 onto the stack
iadd to sum the 1 and 2 on the stack and store the result 3
istore_0 to store the value in the stack into local variable slot 0 -- namely “foo”

Dalvik is Register Based

int foo = 1+2;

		dest	arg1	arg2
0	const-4	%r0	1	
1	add-int/lit8	%r1	%r0	2



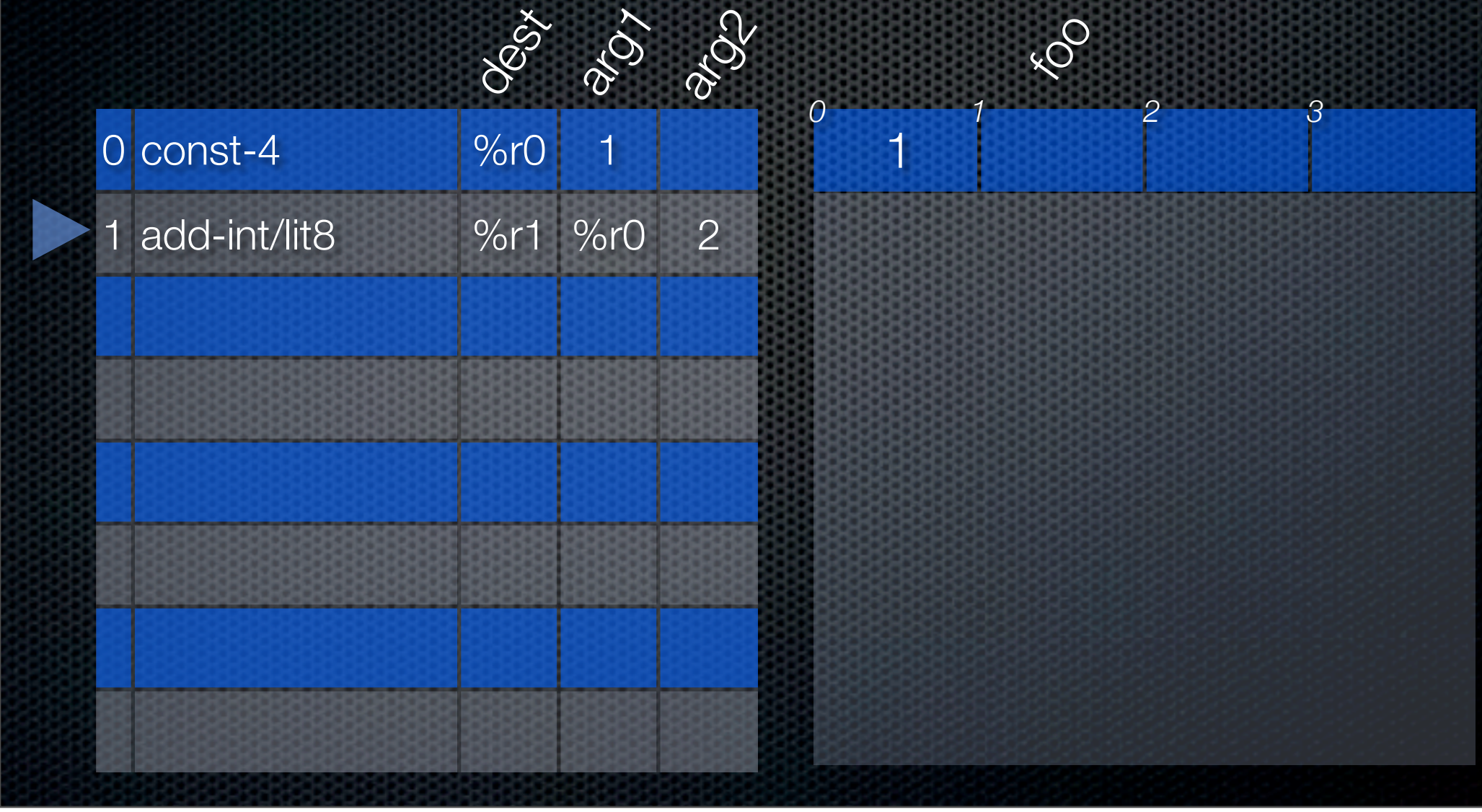
Dalvik bytecode is register based -- it uses 3-operand form -- which it what a processor actually uses
To run “int foo = 1 + 2;”, we run...
const-4 to store 1 into register 0
add-int/lit8 to sum the value in register 0 (1) with the literal 2 and store the result into register 1 -- namely “foo”

This is only 2 dispatches, but Dalvik byte code is measured into 2-byte units
Java byte code was 4-bytes, the Dalvik byte code is actually 6-bytes

However, fewer dispatches generally means less time spent reading code and more time spent running it by the interpreter

Dalvik is Register Based

int foo = 1+2;



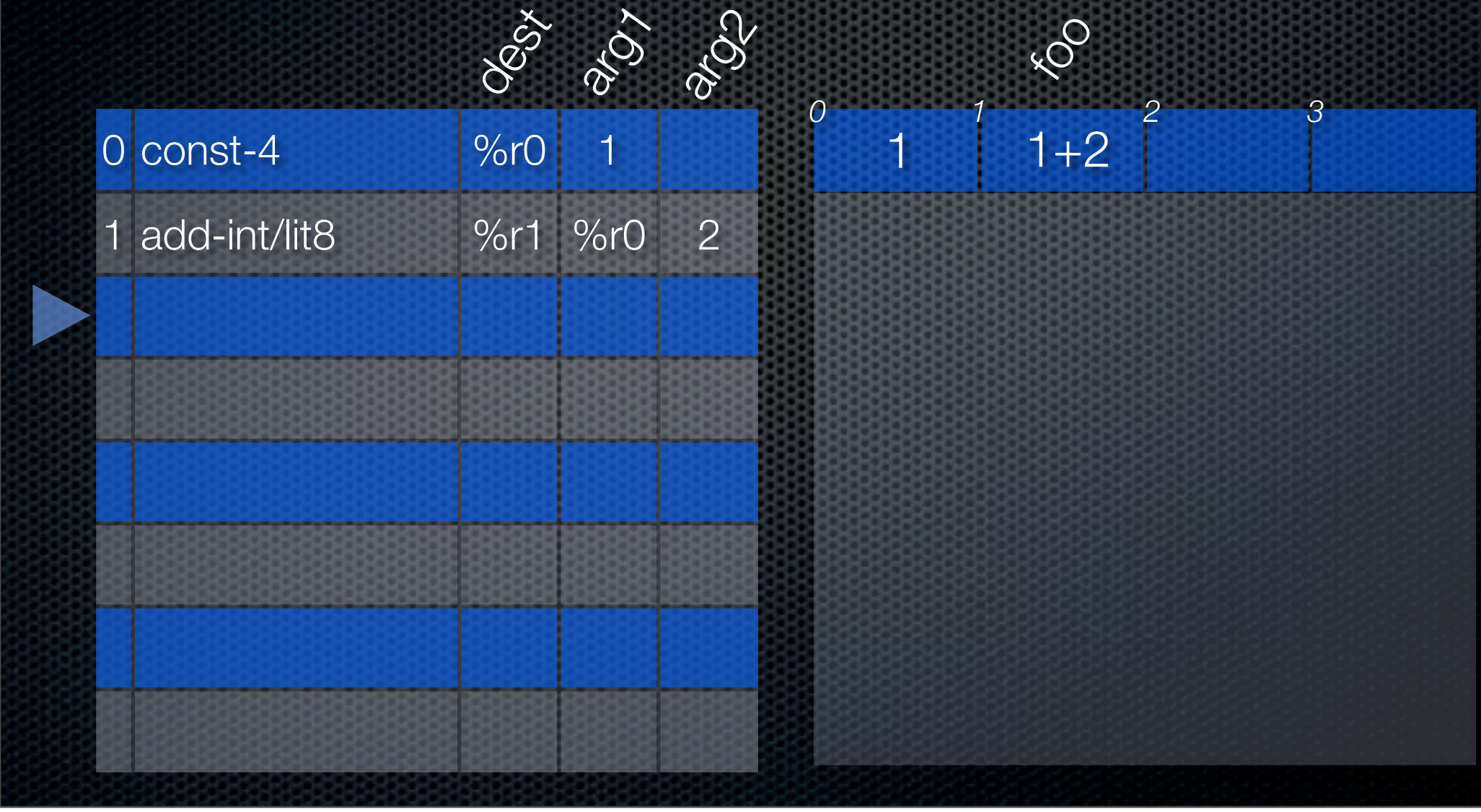
Dalvik bytecode is register based -- it uses 3-operand form -- which it what a processor actually uses
To run “int foo = 1 + 2;”, we run...
const-4 to store 1 into register 0
add-int/lit8 to sum the value in register 0 (1) with the literal 2 and store the result into register 1 -- namely “foo”

This is only 2 dispatches, but Dalvik byte code is measured into 2-byte units
Java byte code was 4-bytes, the Dalvik byte code is actually 6-bytes

However, fewer dispatches generally means less time spent reading code and more time spent running it by the interpreter

Dalvik is Register Based

int foo = 1+2;



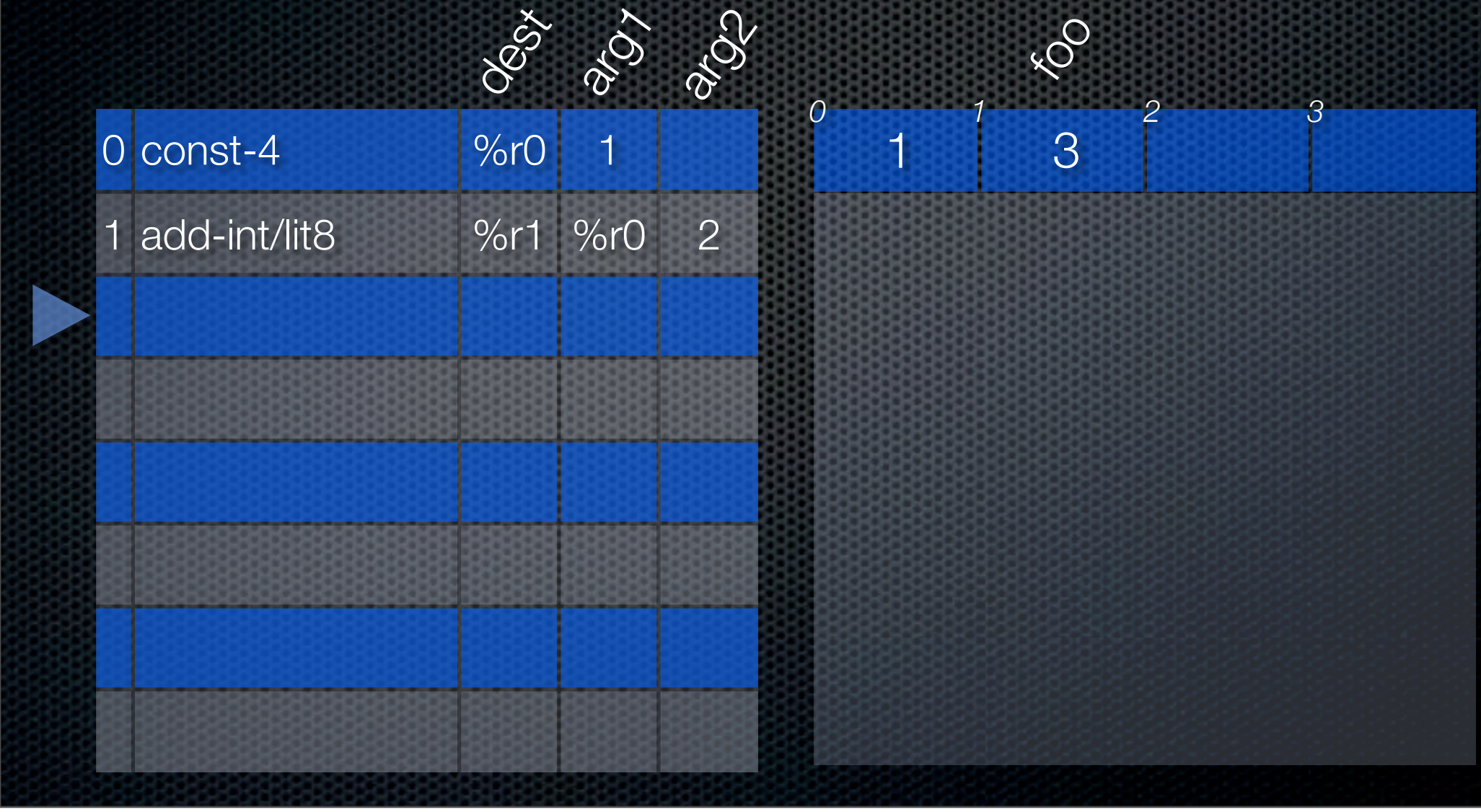
Dalvik bytecode is register based -- it uses 3-operand form -- which it what a processor actually uses
To run “int foo = 1 + 2;”, we run...
const-4 to store 1 into register 0
add-int/lit8 to sum the value in register 0 (1) with the literal 2 and store the result into register 1 -- namely “foo”

This is only 2 dispatches, but Dalvik byte code is measured into 2-byte units
Java byte code was 4-bytes, the Dalvik byte code is actually 6-bytes

However, fewer dispatches generally means less time spent reading code and more time spent running it by the interpreter

Dalvik is Register Based

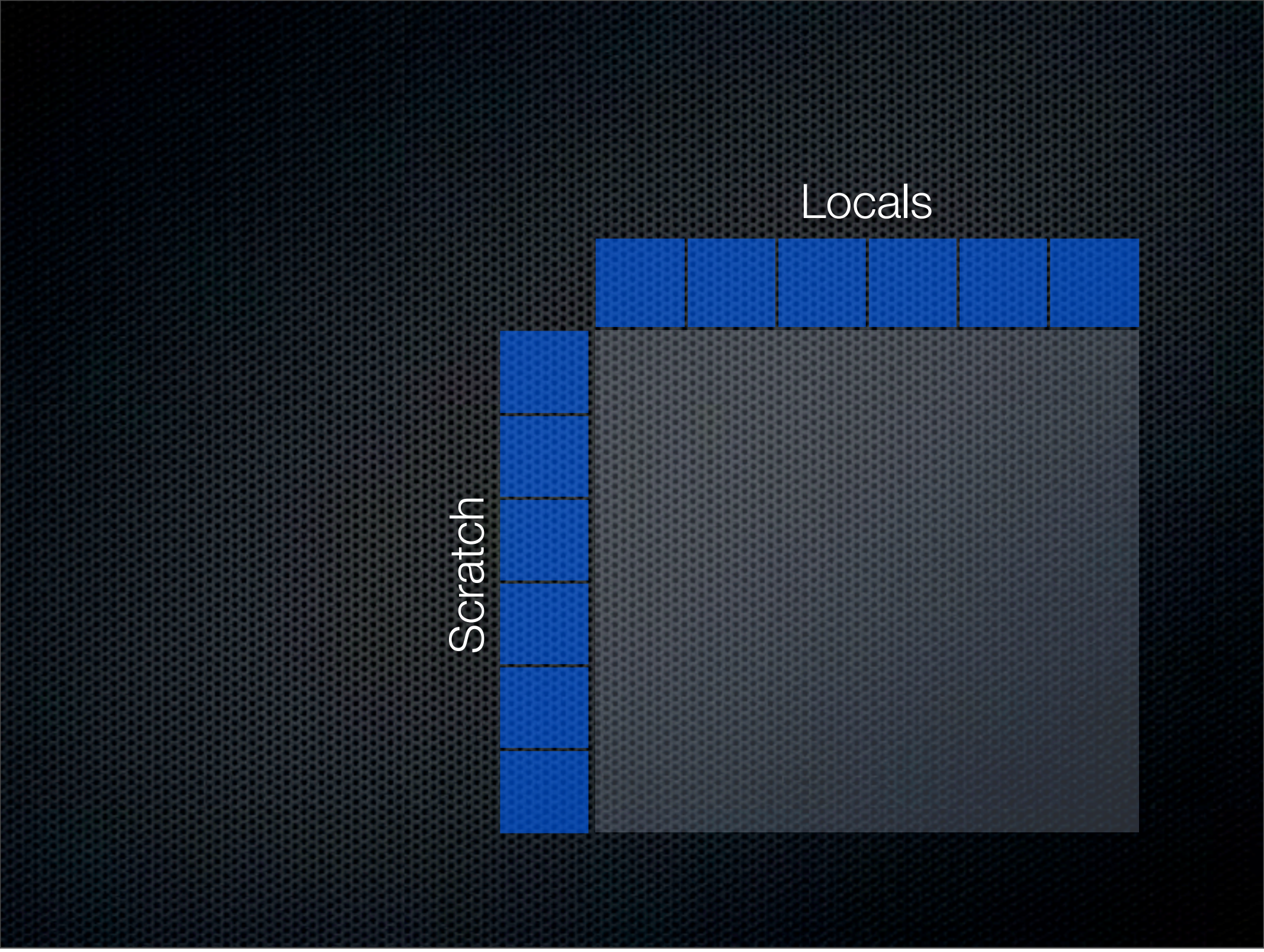
int foo = 1+2;



Dalvik bytecode is register based -- it uses 3-operand form -- which it what a processor actually uses
To run “int foo = 1 + 2;”, we run...
const-4 to store 1 into register 0
add-int/lit8 to sum the value in register 0 (1) with the literal 2 and store the result into register 1 -- namely “foo”

This is only 2 dispatches, but Dalvik byte code is measured into 2-byte units
Java byte code was 4-bytes, the Dalvik byte code is actually 6-bytes

However, fewer dispatches generally means less time spent reading code and more time spent running it by the interpreter

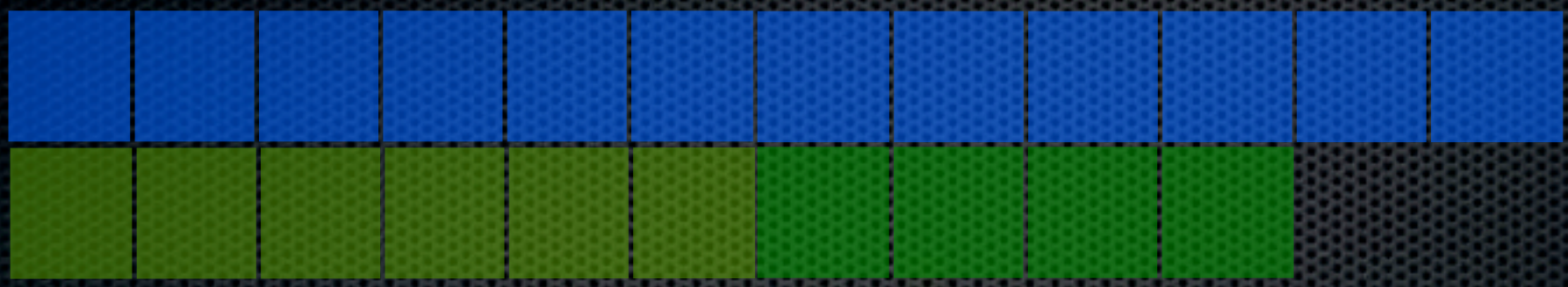


http://www.stanford.edu/class/cs343/resources/java-hotspot.pdf
The way, I'd like you to think about this is...
In a normal VM, local slots are for local variables and stack is scratch space for computation.
In Dalvik, we just treat them all as registers.

After all, in both VMs, they get mapped back to registers in the end.
In fact, JVMs are not as stack-based as they'd lead you to believe.
Consider, the for loop shown here, it is not legal to do just a push of a number onto the stack inside a loop in Java byte code.
Why?
Well, to be able map, stack slots to hardware registers, we need the stack height to be the same at the start and end of a loop -- unlike a true stack-based language like Forth.

The irony is in the end, normal JVMs convert to the same form as Dalvik anyway.
For instance, the HotSpot 6 client JIT works by...

Registers



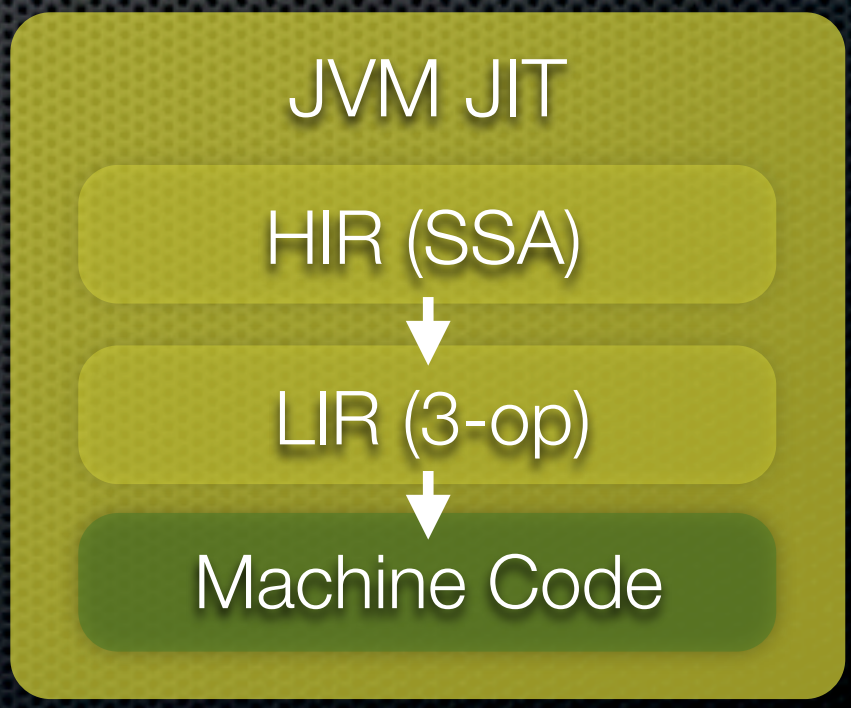
http://www.stanford.edu/class/cs343/resources/java-hotspot.pdf
The way, I'd like you to think about this is...
In a normal VM, local slots are for local variables and stack is scratch space for computation.
In Dalvik, we just treat them all as registers.

After all, in both VMs, they get mapped back to registers in the end.
In fact, JVMs are not as stack-based as they'd lead you to believe.
Consider, the for loop shown here, it is not legal to do just a push of a number onto the stack inside a loop in Java byte code.
Why?
Well, to be able map, stack slots to hardware registers, we need the stack height to be the same at the start and end of a loop -- unlike a true stack-based language like Forth.

The irony is in the end, normal JVMs convert to the same form as Dalvik anyway.
For instance, the HotSpot 6 client JIT works by...

Registers


```
for (int i = 0; i < 100; ++i ) {
  push(10);
}
```



http://www.stanford.edu/class/cs343/resources/java-hotspot.pdf

The way, I'd like you to think about this is...
In a normal VM, local slots are for local variables and stack is scratch space for computation.
In Dalvik, we just treat them all as registers.

After all, in both VMs, they get mapped back to registers in the end.
In fact, JVMs are not as stack-based as they'd lead you to believe.
Consider, the for loop shown here, it is not legal to do just a push of a number onto the stack inside a loop in Java byte code.
Why?
Well, to be able map, stack slots to hardware registers, we need the stack height to be the same at the start and end of a loop -- unlike a true stack-based language like Forth.

The irony is in the end, normal JVMs convert to the same form as Dalvik anyway.
For instance, the HotSpot 6 client JIT works by...

JVM Types vs Dalvik Types

int float pointers	Category1	Normal
long double	Category2	Wide

The JVM broadly divides types into two categories: 1 & 2
Category 1 types are 32-bit -- these are int, float, and pointers to Objects
Category 2 types are 64-bit -- these are long and double
What about the smaller types: short, byte, char, boolean? -- well in registers they are just ints

All slots / registers are 32-bit in a normal JVM and Dalvik...
So, 64-bit types take up two registers

Dalvik uses normal and wide to describe these groups, but otherwise things are mostly the same.

As an aside, the JVM bytecode treats pointers as 32-bits, but the JVM doesn't always.
A 64-bit JVM may treat them as 64-bit (i.e. Category 2) -- except when it doesn't because it uses compressed pointers and puts 2 x 32-bit pointers in a single 64-bit register

JVM Bytecode *vs* Dalvik Bytecode

nop	
ifeq / ifne	2-byte
iflt / ifle	2-byte
ifgt / ifge	2-byte
ifnull	2-byte
ifnonnull	2-byte

nop	byte	
if-eqz / if-nez	byte	2-byte
if-ltz / if-lez	byte	2-byte
if-gtz / if-gez	byte	2-byte

Let's compare some bytecode instructions side-by-side...

In many cases, things are mostly the same...
Regular JVM has a nop -- so does Dalvik, but Dalvik's takes an extra byte (which is unused)
Sacrifices space, but makes the interpreter more efficient

Consider the instructions, that compare to zero and jump
Dalvik's have a hyphen and z at the end, but otherwise they are the same
They do take an extra argument to specify the register to compare against, but that's it.

However, Dalvik's if-eqz and if-nez are overloaded. The regular JVM has separate ifnull and ifnonnull instructions that Dalvik lacks -- it justs if-eqz and if-nez respectively.

JVM Bytecode *vs* Dalvik Bytecode

iconst_m1 - 5	
bipush	byte
sipush	2-byte
ldc	byte
istore_0 - 3	
istore	byte
iload_0 - 3	
iload	byte

const4	nibble	nibble
const16	byte	2-byte
const/high16	byte	2-byte
const	byte	4-byte
move	byte	byte
move/from16	byte	2-byte
move/16	2-byte	2-byte

Continuing the comparison, let's look at loading a number into a slot

JVM bytecode has special instructions inconst_m5 through iconst_5 for values: -1 to 5.
To load into a local variable slot, we'd follow it up with a istore_0 - 3 or a plain istore.
Note, istore takes a byte which indicates that the regular JVM only has 256 local variable slots.

So, in the end for the common cases: -1 to 5 loaded into slots: 0-3, we use two bytes and two dispatches.

On Dalvik, we can use const4 which takes a nibble for the destination register and a nibble for the value.
Worth noting, the nibble for the value is signed so it covers all values -8 to 7.
Also, it can load into register 0-15.

For large values, JVM has ldc which takes a byte, but it is an index into the constant

JVM Bytecode *vs* Dalvik Bytecode

getfield	2-byte

iget	nib	nib	2-byte
iget-wide	nib	nib	2-byte
iget-object	nib	nib	2-byte
iget-short	nib	nib	2-byte
iget-char	nib	nib	2-byte
iget-byte	nib	nib	2-byte
iget-boolean	nib	nib	2-byte

Looking at types, a little more sometimes Android is a lot more granular than the JVM.
When loading from an instance field, the JVM has a single instruction: getfield.
Android separates these into iget (for normal), iget-wide, and iget-object, but...
It also has instructions for all the smaller types, too.

JVM Bytecode

vs

Dalvik Bytecode

iastore	
fastore	
lastore	
dastore	
aastore	
sastore	
castore	
bastore	

aget	nib	nib	2-byte
aget-wide	nib	nib	2-byte
aget-object	nib	nib	2-byte
aget-short	nib	nib	2-byte
aget-char	nib	nib	2-byte
aget-byte	nib	nib	2-byte
aget-boolean	nib	nib	2-byte

...but, sometimes, it goes the other way. The JVM makes a distinction ints and floats when dealing with arrays, but Android just has aget.
Similarly, for wide types.
They both have instructions for objects, shorts, chars, and bytes, but the regular JVM does not have an instruction for booleans. In the regular JVM, boolean arrays are manipulated using byte instructions.

Fewer Instructions / Dispatches
Fewer Reads
Fewer Writes

Closer to Machine Code

The net result...
Byte code is a little bigger, but we do fewer dispatches
We also (at least at the bytecode level), do fewer reads and fewer writes
Also, the bytecode is closer to machine code making the translation a little easier

In the end, the Dalvik interpreter is about twice as efficient as a normal JVM interpreter.
Of course, that's no real accomplishment, a JIT is orders of magnitude faster in the end.

Demo

Compiler API + dex + baksmali

http://code.google.com/p/smali/

Now, we're going to look at some Java examples -- shown side-by-side in both Java and Dalvik

For the ambitious, if you set-up the accompanying Eclipse project and run Serve.

You can use the interactive web-app by directing your browser to localhost:8080/compile.

These demos are built on-top of Java 6's compiler API, Android's dex tool, and DEX disassembler named baksmali (another Icelandic reference).

Hello World

System.out.println("Hello World");

JVM Bytecode

```
getstatic System.out
ldc "Hello World"
invokevirtual
  PrintStream.println
```

Dalvik Bytecode

```
sget-object
  v0, System.out
const-string
  v1, "Hello World"
invoke-virtual
  {v0, v1},
  PrintStream.println
```


First, let's start -- where we always start "Hello World"
Both use 3 dispatches -- both use same number of reads and writes
Both jump to constant pool for the string

Array

int[] xs = {20, 30, 0, 50};

JVM Bytecode

```
iconst_4 // 0x04
newarray int
dup
iconst_0 // 0x00
bipush 0x14
iastore
dup
iconst_1 // 0x01
iastore
dup
iconst_3 // 0x03
bipush 0x32
iastore
astore_1 // 01
```

Dalvik Bytecode

```
const/4 v0, 0x04
new-array v0, v0, [I
fill-array-data
  v0, :array_8
nop
:array_8
0x14 0x00 0x00 0x00
0x1e 0x00 0x00 0x00
0x00 0x00 0x00 0x00
0x32 0x00 0x00 0x00
```


http://www.youtube.com/watch?v=ptjedOZEXPM

Arrays are particularly inefficient in Java, consider the example above...

For every element, Java adds a dup, an index load, a value push onto the stack, and an array store.

At best, 4 bytes per element. bipush - immediately pushes it to 5 and gets worse from there.

For large ints, have to jump to constant pool for the value which provides poor cache usage.

Android simply has fill-array-data which takes an offset. Data is held close to the code for better cache utilization. And, that extra nop that's to align the data on a 4-byte boundary to make the load that much more efficient.

Invoke Static

Math.max(10, 20);

JVM Bytecode

```
bipush 0x0a
bipush 0x14
invokestatic Math.max
pop
```

Dalvik Bytecode

```
const/16 v0, 0x0a
const/16 v1, 0x14
invoke-static
  {v0, v1}, Math.max
```


Looking at invocation of a static method...
Basically, the same - 3 dispatches, but JVM has extra pop at the end
Why, well, the return value of max which was placed on to the stack has to be consumed somehow, so the JVM adds an extra pop.
Actually, Dalivk does the same thing if you disable optimization, you'll see a move-result in the Dalvik bytecode.
However, because this is just a special register, there's no harm in not moving it, so Dalvik just gets rid of it.
This highlights another interesting difference Dalvik does compile time optimization.
The regular JVM tool chain used to do this pre-HotSpot, but starting with HotSpot (Java 1.3) optimization purely became the job of the VM.
Since Android uses an interpreter, static optimizations are still useful.

New

BigDecimal x = new BigDecimal("2.0");

JVM Bytecode

```
new BigDecimal
dup
ldc "2.0"
invokespecial
  BigDecimal.<init>
astore_1 // 01
```

Dalvik Bytecode

```
new-instance
  v0, BigDecimal;
const-string
  v1, "2.0"
invoke-direct
  {v0, v1},
  BigDecimal.<init>
```


Finally, allocating a new Object.
Allocating a new Object is rather ugly in normal bytecode.
First we allocate a raw slab of memory for the object and place a reference on the stack.
The constructor has not been invoked yet and invoking will consume the reference on the stack, so we “dup” the reference.
Then invoke the constructor, consuming one reference -- leaving one left that we can store into a local variable slot.

In Dalvik, it is easier because everything is just a register. The reference we put into the slot in dispatch 1 is still there when we do the invoke in dispatch 3, so no need for a “dup”.

JVM has 1w / 1r + 1w / 1w / 2r+1w / 1w -> 3r + 5w
Dalvik has 1w / 1w / 2r + 1w -> 2r + 3w

Efficient Interpreter

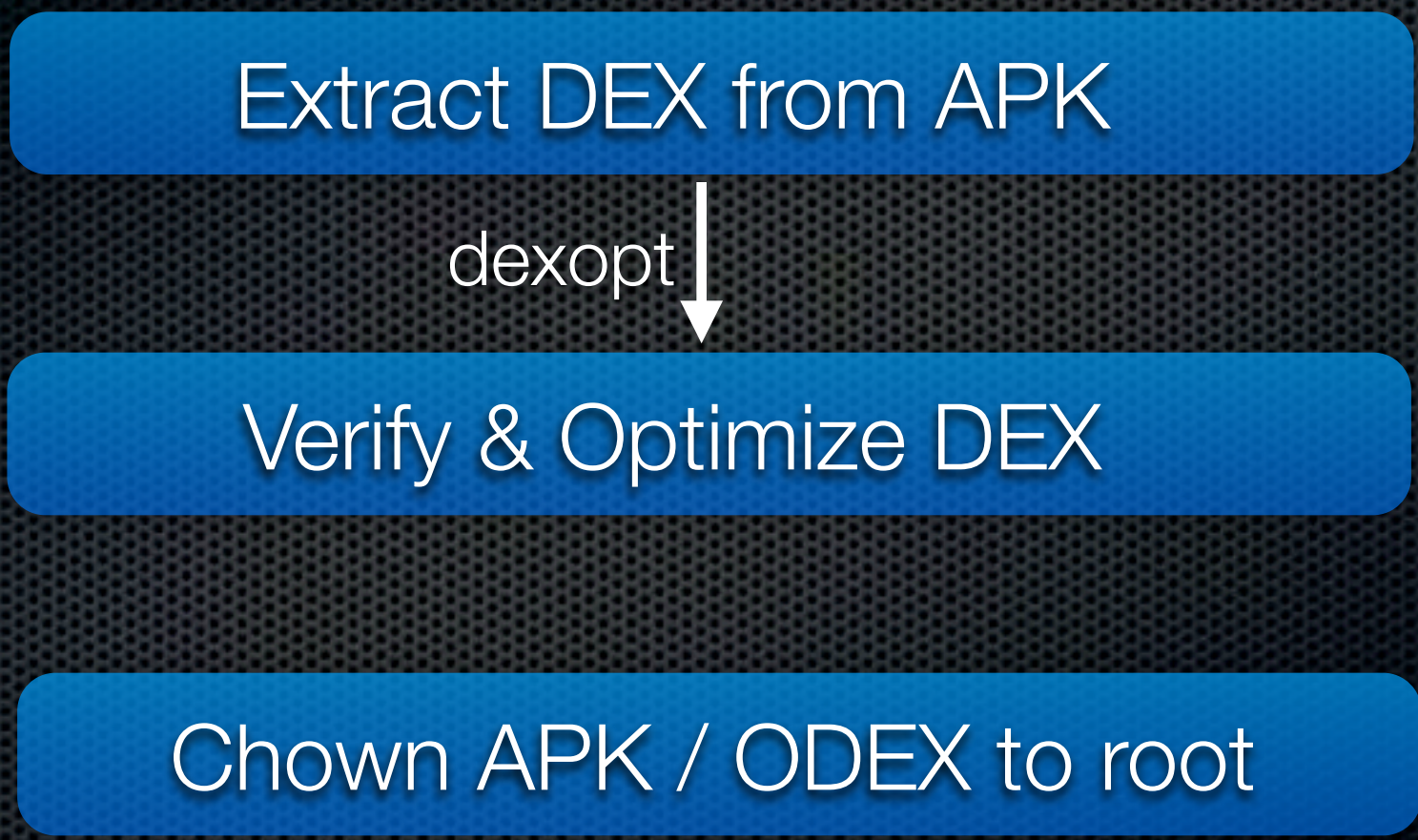
3rd form of Dalvik

Zygote Dalvik DexOpt

ODEX = Optimized DEX

The final form of Dalvik -> dexopt - used at install time to make interpreter even more efficient

Installation Process



http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html
Not only does Android optimize at translation time, it also optimizes at install time
When the APK is installed, the DEX is extracted from the APK and placed in /data/dalvik-cache
But it is also optimized -- and optimized to the particular phone
If the file is big endian and the phone is little endian, the endianness is switched
Locking will be handled differently for SMP and non-SMP systems
...and some bytecode optimizations

This addresses performance and launch speed, since verification is only down at install.

More on verification: http://www.milk.com/kodebase/dalvik-docs-mirror/docs/verifier.html

Optimized DEX

invoke-virtual	nib	2-byte
java/lang/String#length():I		

execute-inline	nib	nib
java/lang/String#length():I		

invoke-direct	nib	2-byte
java/lang/Object#<init>():V		

http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html

Let's look at just a couple optimizations

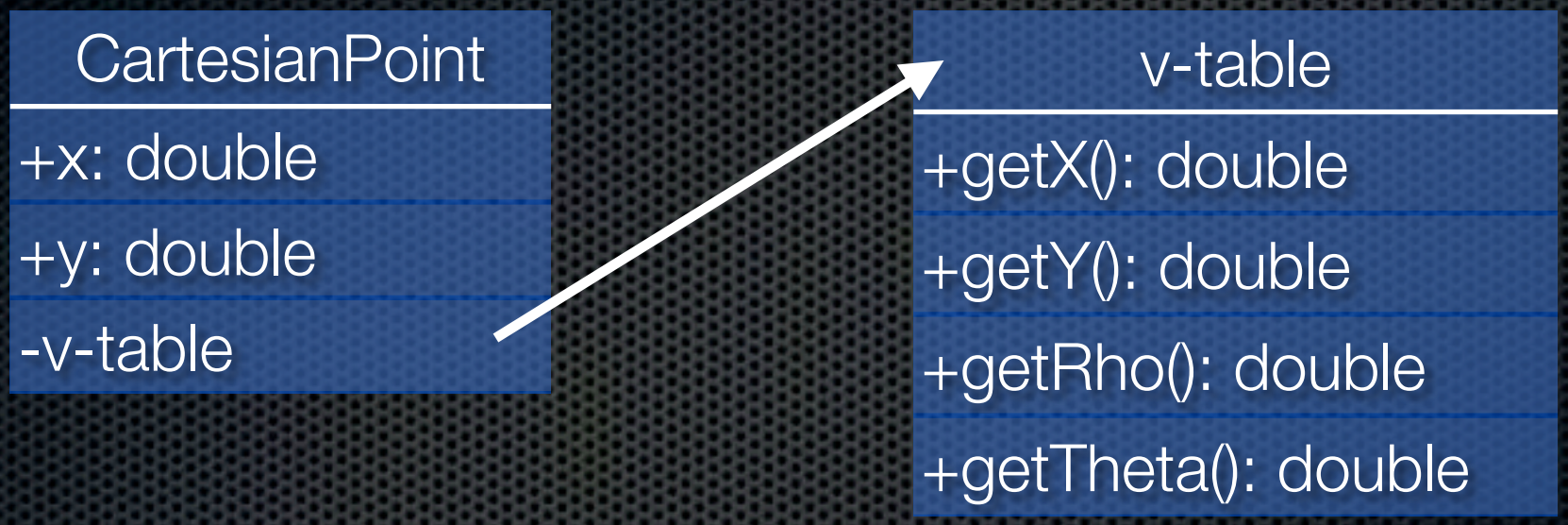
First, common operations like String.length have their own special instruction execute-inline

VM has special code just for those common operations

Things like calling the Object's constructor - optimized to nothing because the method is empty

http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html

Optimized DEX



iget	nib	nib	2-byte	iget-quick	nib	nib	2-byte

invoke-virtual	nib	2-byte	invoke-virtual-quick	nib	2-byte

http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html

Can also do other things because we know how object will be laid out in memory

Can change a field iget to iget-quick which does a simple pointer bump

Can change invoke-virtual to invoke-virtual-quick because we know the layout of the v-table

http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html

Demo

DEX-es

> ls /data/dalvik-cache

```
-rw-r--r-- root     root         5857 2011-11-06 17:29 system@app@Maps.apk@classes.dex
-rw-r--r-- root     root       153432 2011-11-06 17:23 system@app@Mms.apk@classes.dex
```


DEX-es are extracted from APKs and placed in /data/dalvik-cache - owned by root, but readable by all

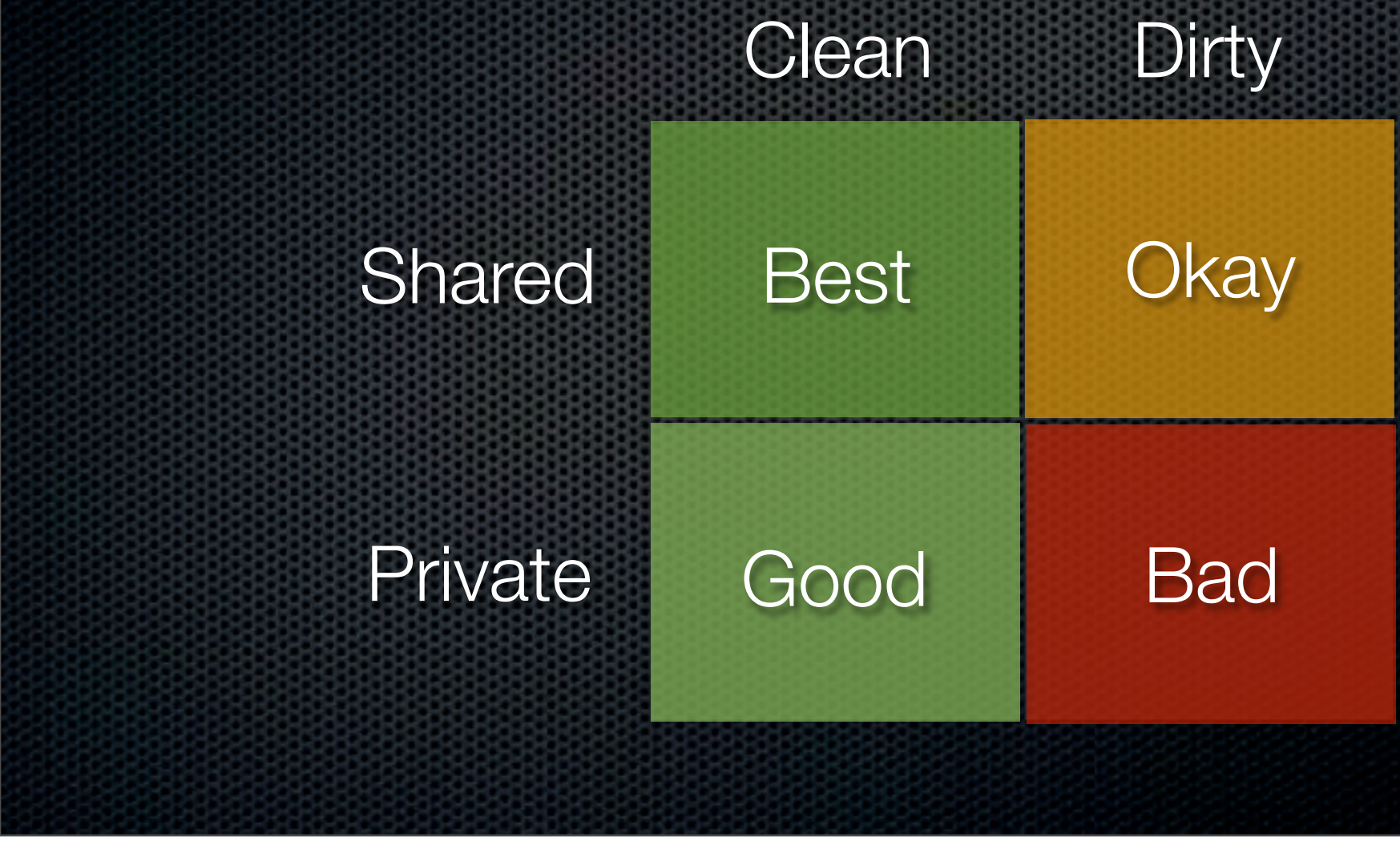
Device Image ODEX-es

```
> ls /system/app
-rw-r--r-- root     root         5857 2011-11-06 17:29 GoogleBackupTransport.apk
-rw-r--r-- root     root       153432 2011-11-06 17:23 GoogleBackupTransport.odex
-rw-r--r-- root     root         8575 2011-11-06 17:29 GoogleCalendarSyncAdapter.apk
-rw-r--r-- root     root       270752 2011-11-06 17:23 GoogleCalendarSyncAdapter.odex
-rw-r--r-- root     root         8563 2011-11-06 17:29 GoogleContactsSyncAdapter.apk
-rw-r--r-- root     root       321552 2011-11-06 17:23 GoogleContactsSyncAdapter.odex
```


Image ODEX-es are stored next to APK in /system/app
These are generated when the image is being built

JIT Compiler

Added in Android 2.2



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

JIT added in Android 2.2

Why? - hard to make a JIT that does not use private dirty memory - the worse kind

So, memory memory use by using a trace JIT

Also, trace JIT produces performance boost faster

JIT Compiler
Added in Android 2.2

Trace - not Method

Memory Pressure

http://www.youtube.com/watch?v=Ls0tM-c4Vfo

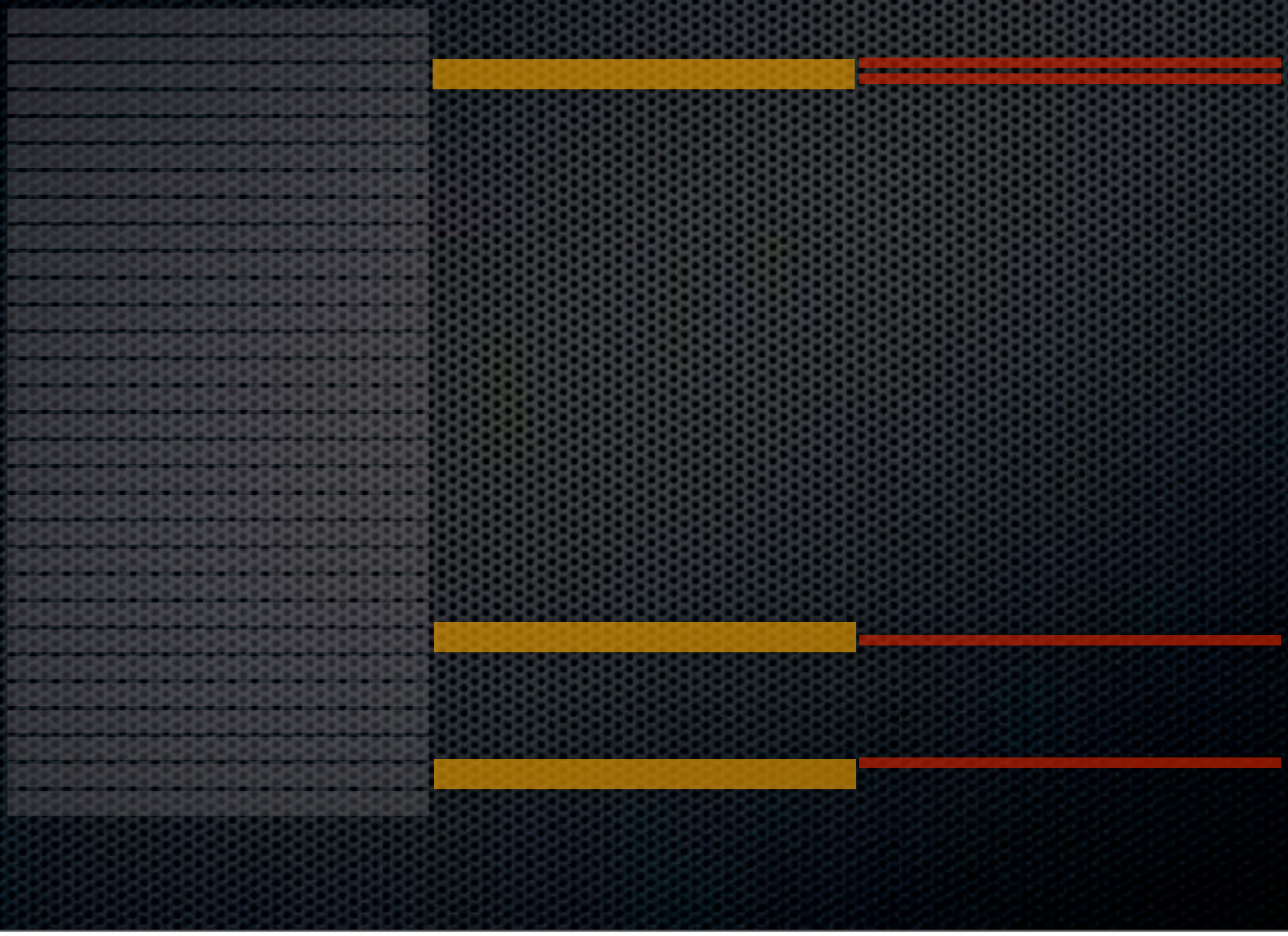
JIT added in Android 2.2

Why? - hard to make a JIT that does not use private dirty memory - the worse kind

So, memory memory use by using a trace JIT

Also, trace JIT produces performance boost faster

Why a Trace JIT?



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

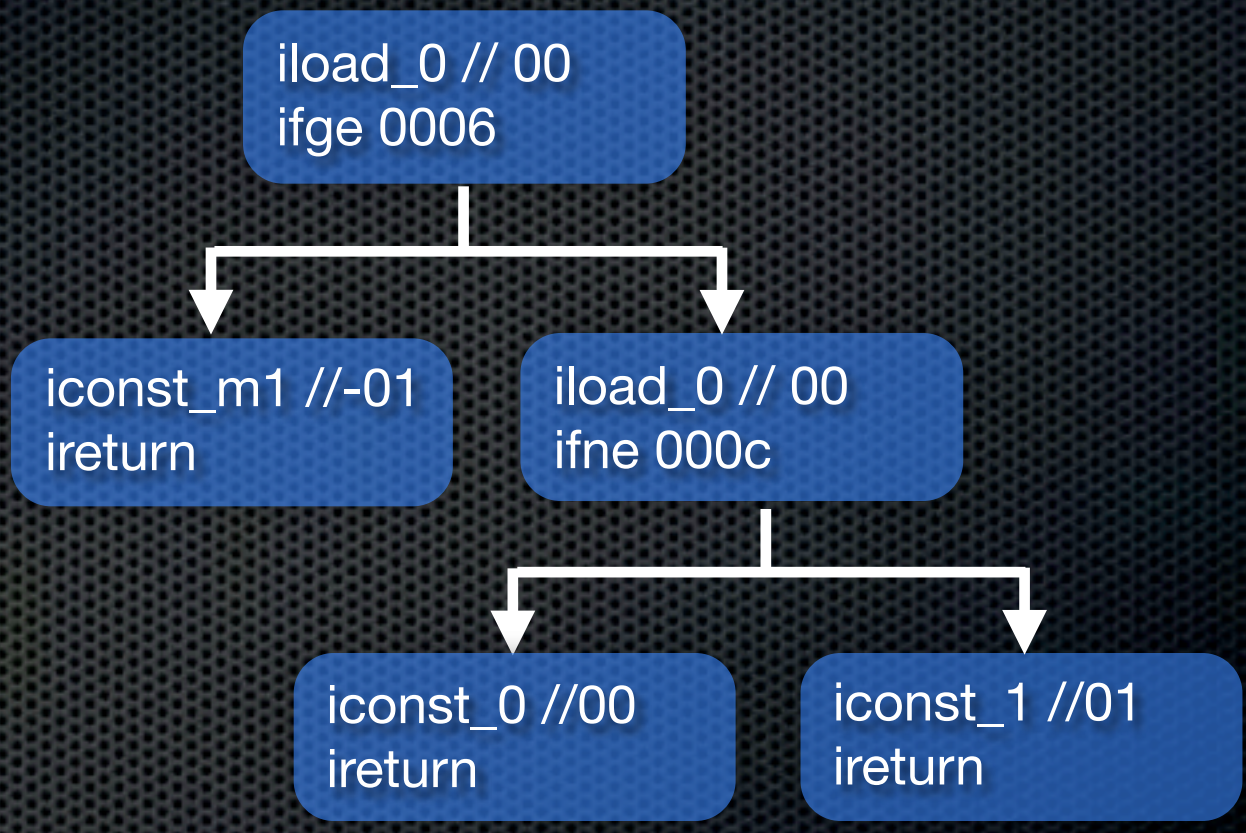
Compiled Code takes up memory - want the benefits of JIT with small memory footprint

Small amount compilation provides a big benefit

In test program, 4.5MB of byte code - 8% of methods: 390K was hot; 25% of code in methods was hot - so 2% in the end

90% of time in 10% of the code may be generous

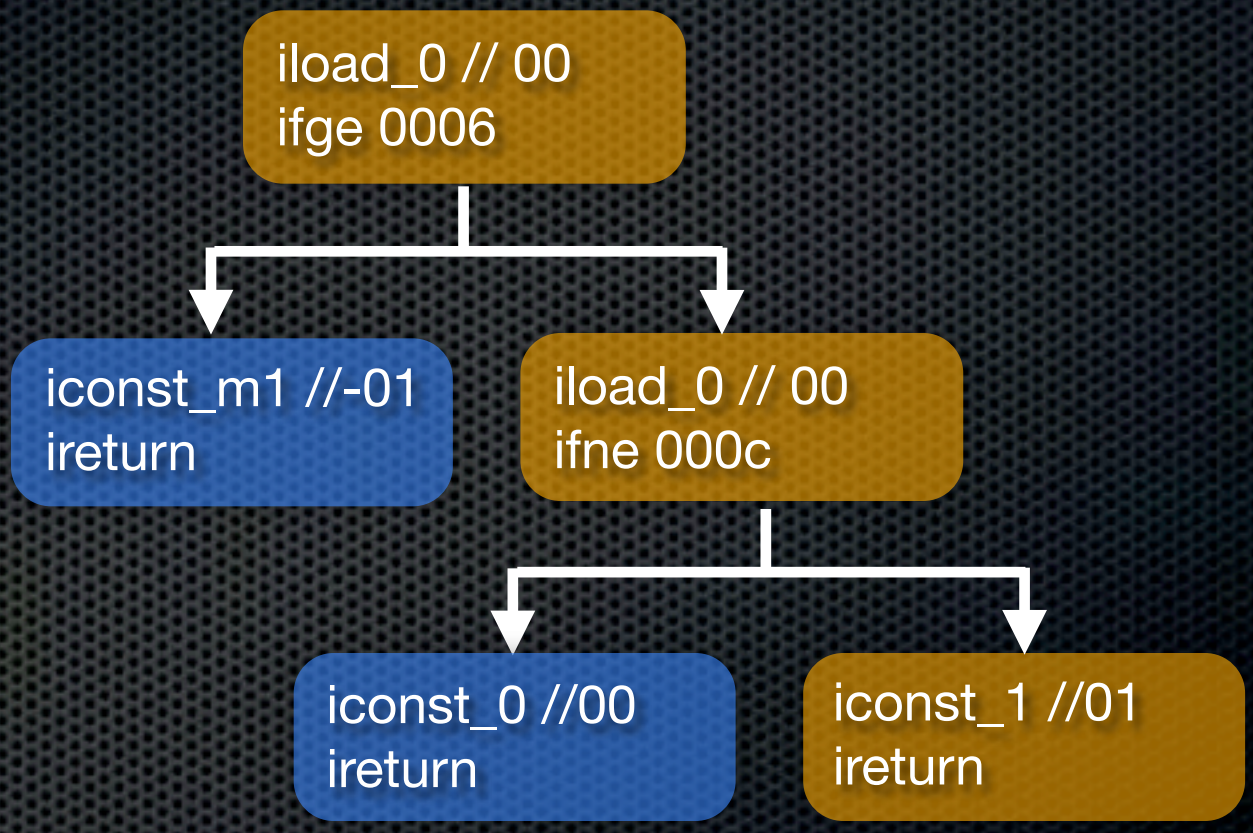
```
int signum( int x ) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return 1;
  }
}
```



http://www.youtube.com/watch?v=Ls0tM-c4Vfo
Identify the hot blocks
Chain together into a linear code segment
Violating constraints bail to the interpreter

These bails to interpreter are called side exits and exist in all VMs. Since almost every instruction can potentially raise an exception, treated exceptions at block boundaries would prevent most useful optimizations. To get around this exceptions are treated as side exits.

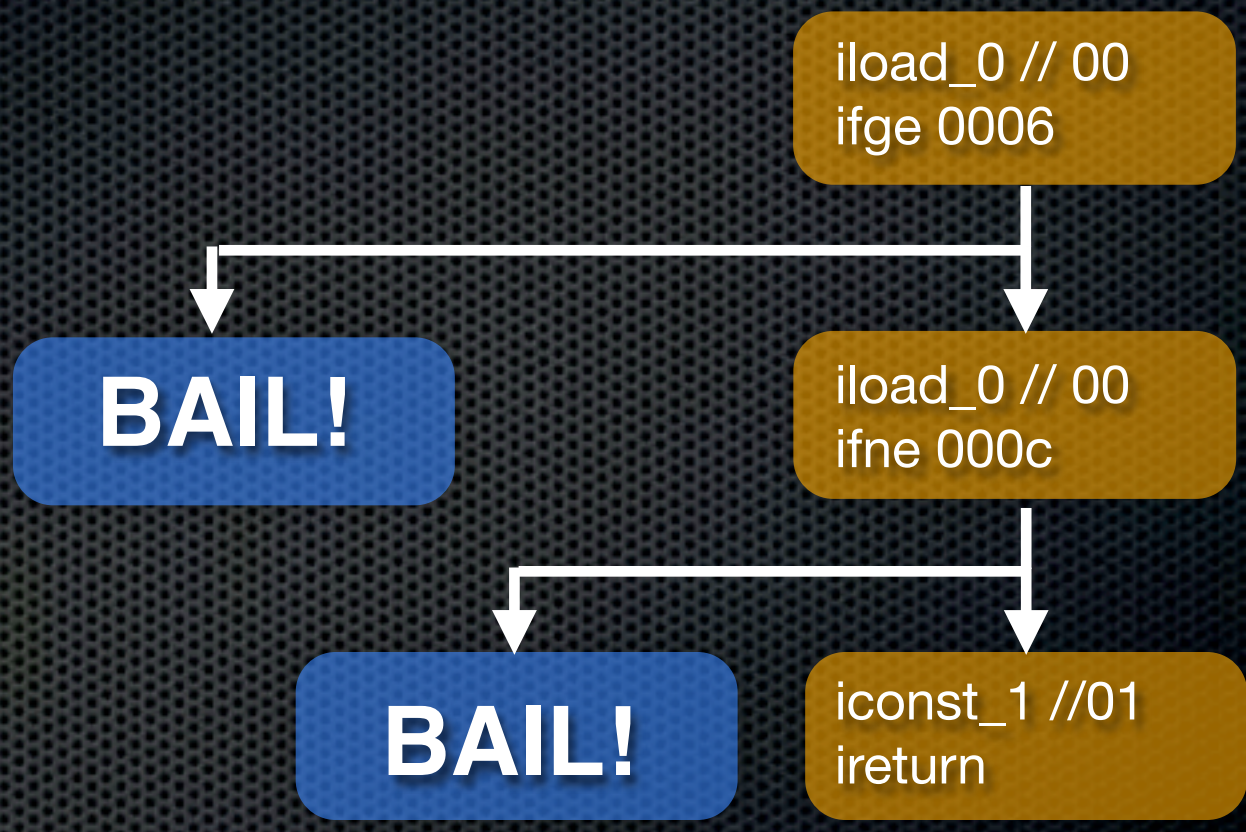
```
int signum( int x ) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return 1;
  }
}
```



http://www.youtube.com/watch?v=Ls0tM-c4Vfo
Identify the hot blocks
Chain together into a linear code segment
Violating constraints bail to the interpreter

These bails to interpreter are called side exits and exist in all VMs. Since almost every instruction can potentially raise an exception, treated exceptions at block boundaries would prevent most useful optimizations. To get around this exceptions are treated as side exits.

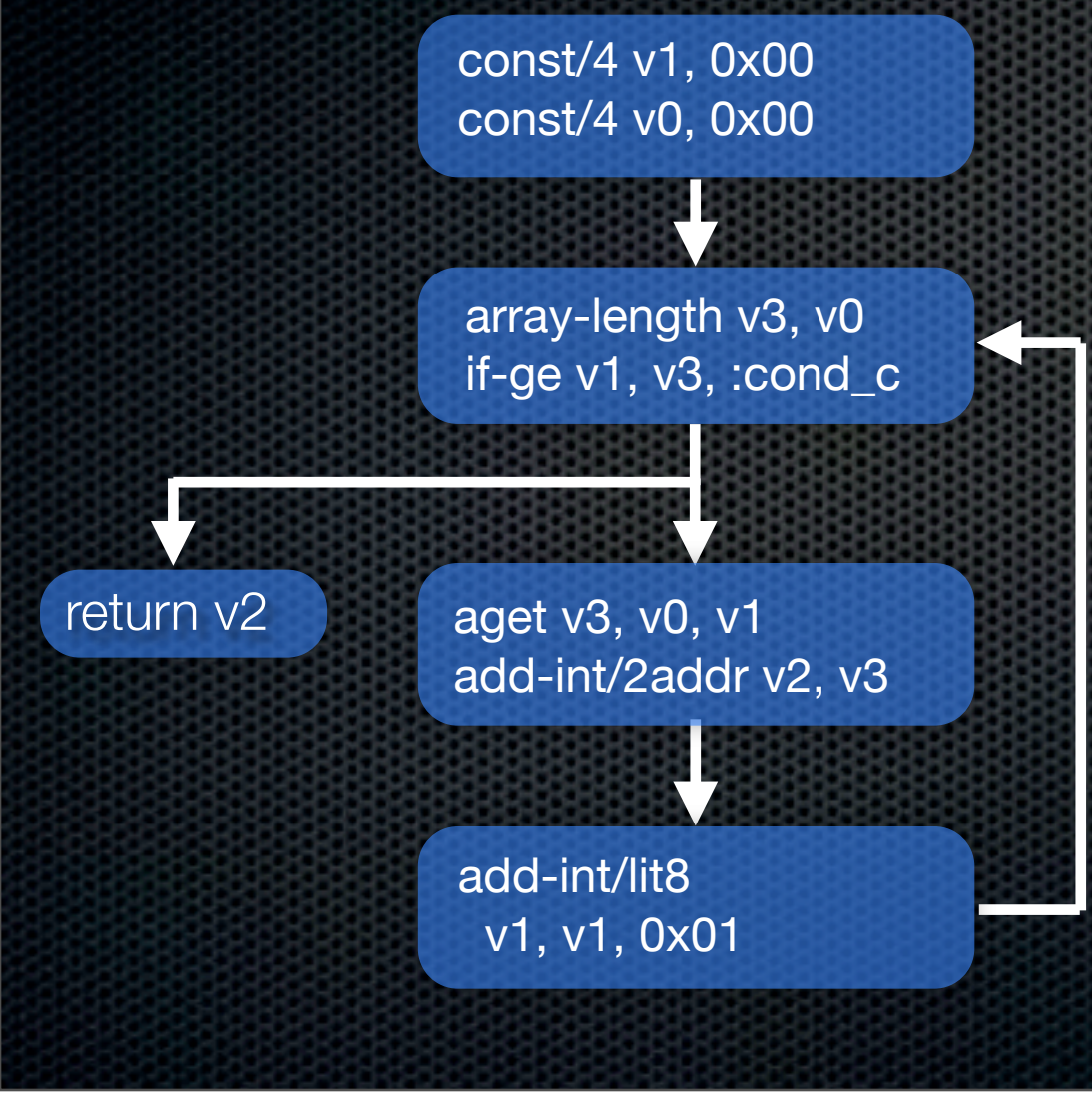
```
int signum( int x ) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return 1;
  }
}
```



http://www.youtube.com/watch?v=Ls0tM-c4Vfo
Identify the hot blocks
Chain together into a linear code segment
Violating constraints bail to the interpreter

These bails to interpreter are called side exits and exist in all VMs. Since almost every instruction can potentially raise an exception, treated exceptions at block boundaries would prevent most useful optimizations. To get around this exceptions are treated as side exits.

```
int sum = 0;
for (int i=0; i<array.length; ++i) {
  sum += array[i];
}
return sum;
```



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

Keep count for each trace head (i.e. first line of block)

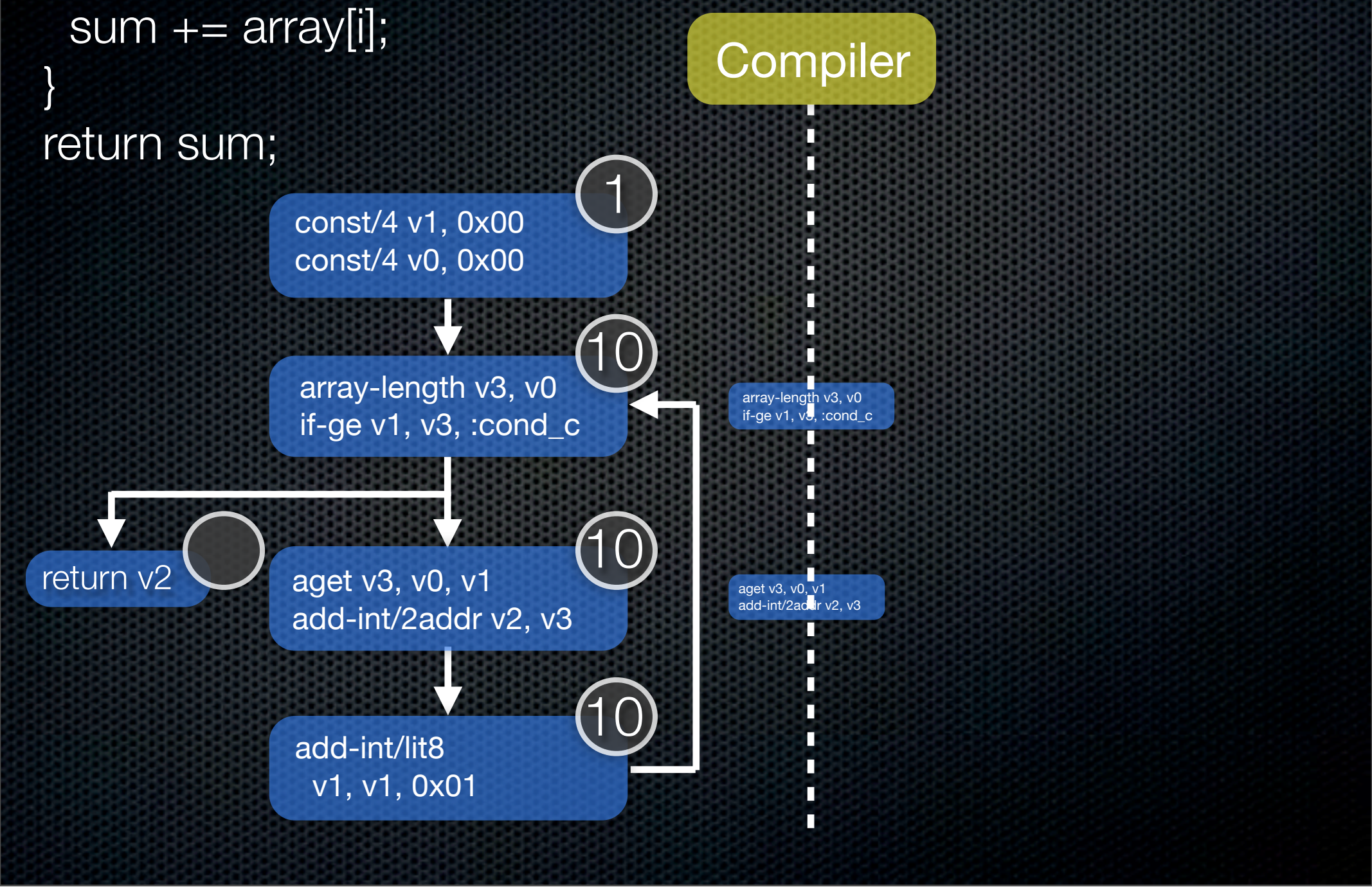
If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted

```
int sum = 0;
for (int i=0; i<array.length; ++i) {
  sum += array[i];
}
return sum;
```



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

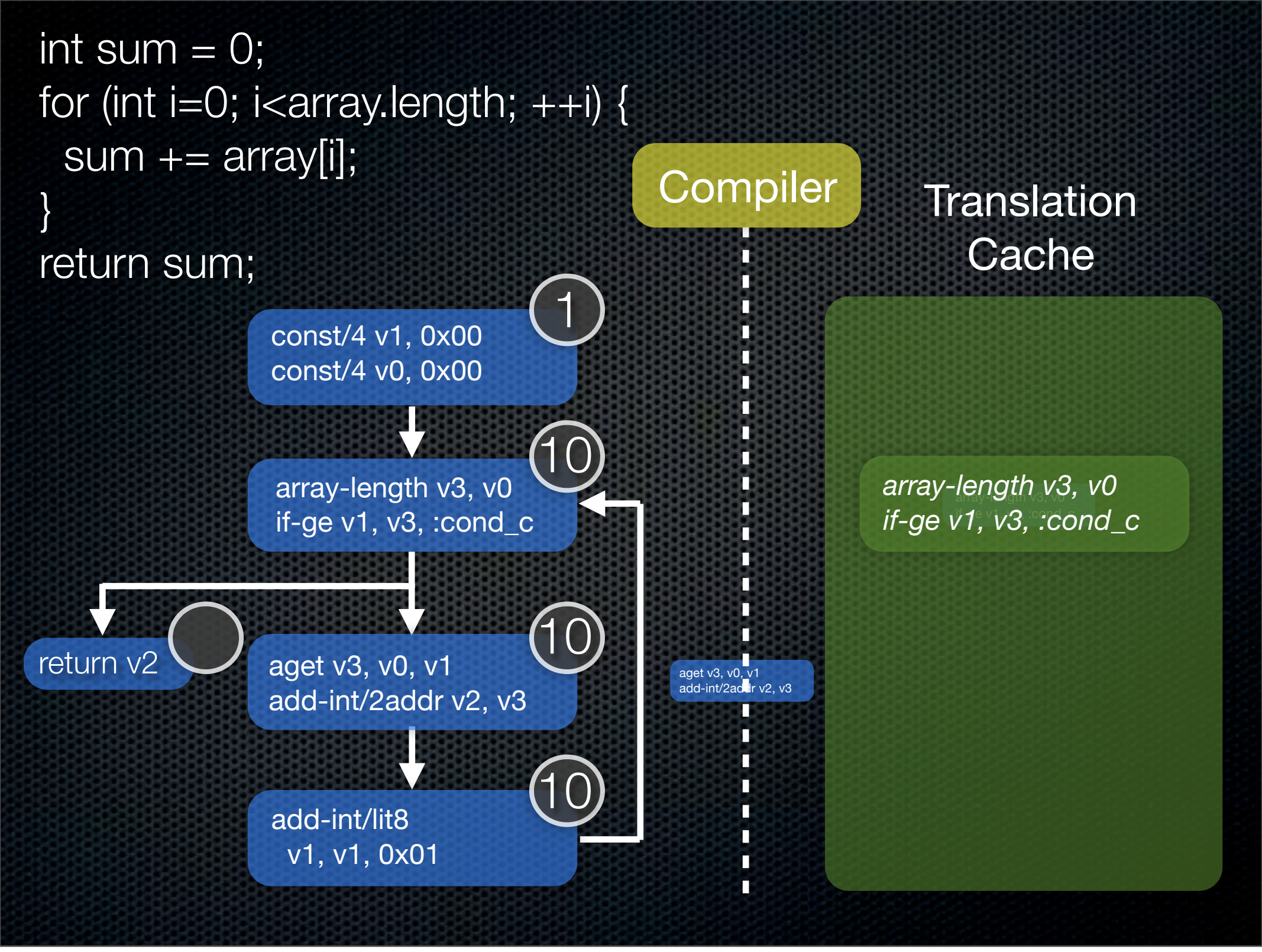
Keep count for each trace head (i.e. first line of block)

If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

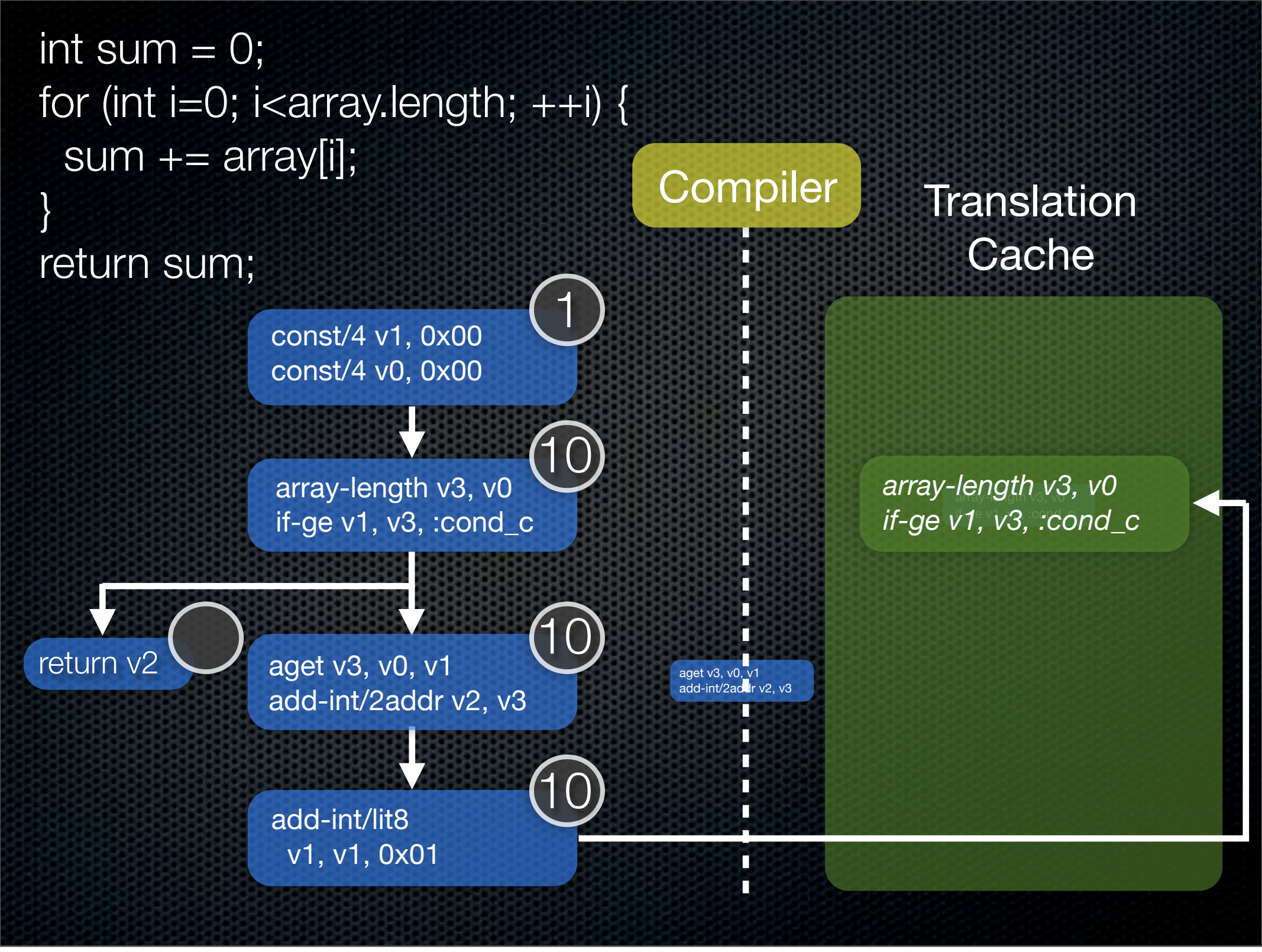
Keep count for each trace head (i.e. first line of block)

If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

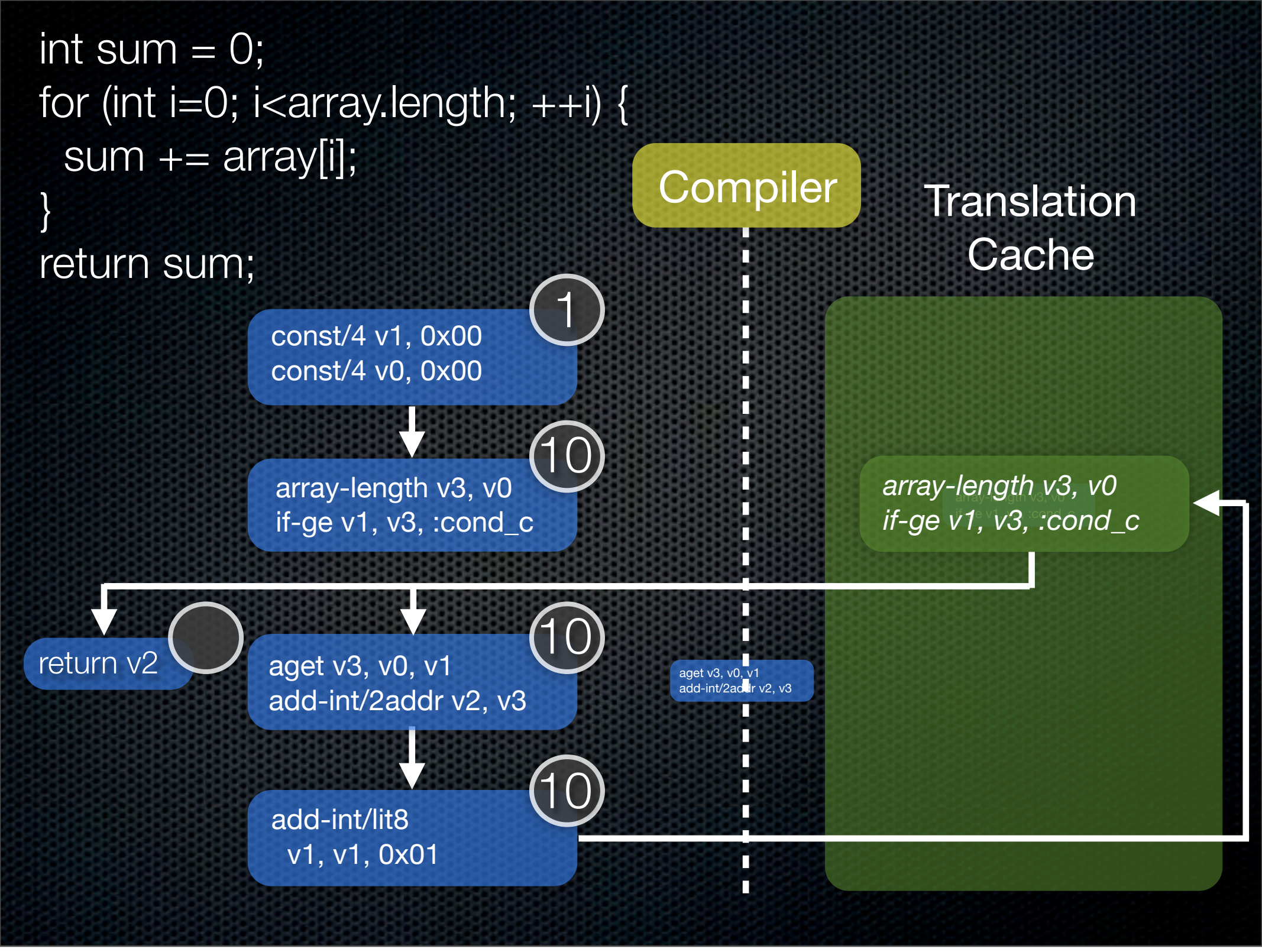
Keep count for each trace head (i.e. first line of block)

If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

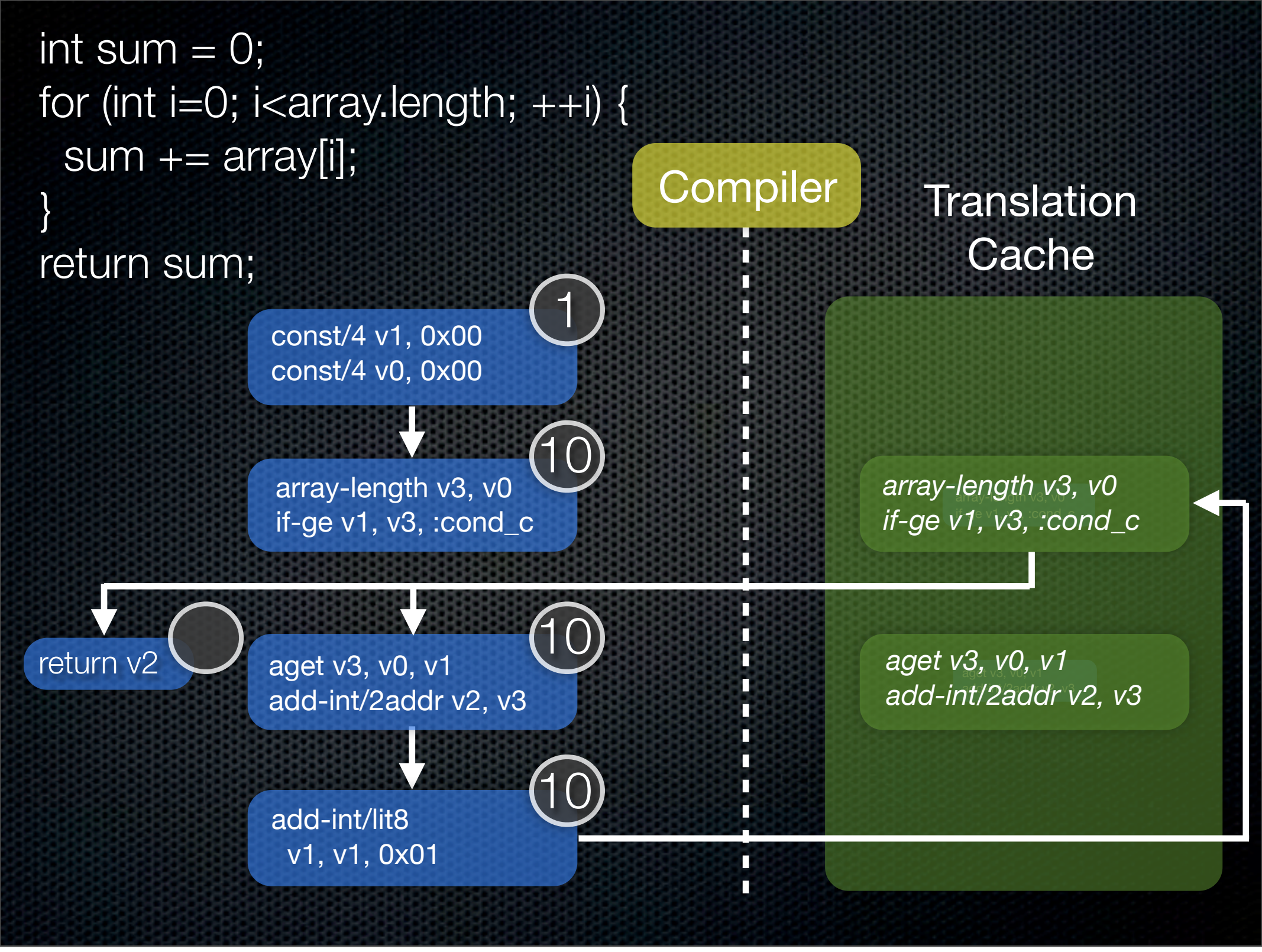
Keep count for each trace head (i.e. first line of block)

If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

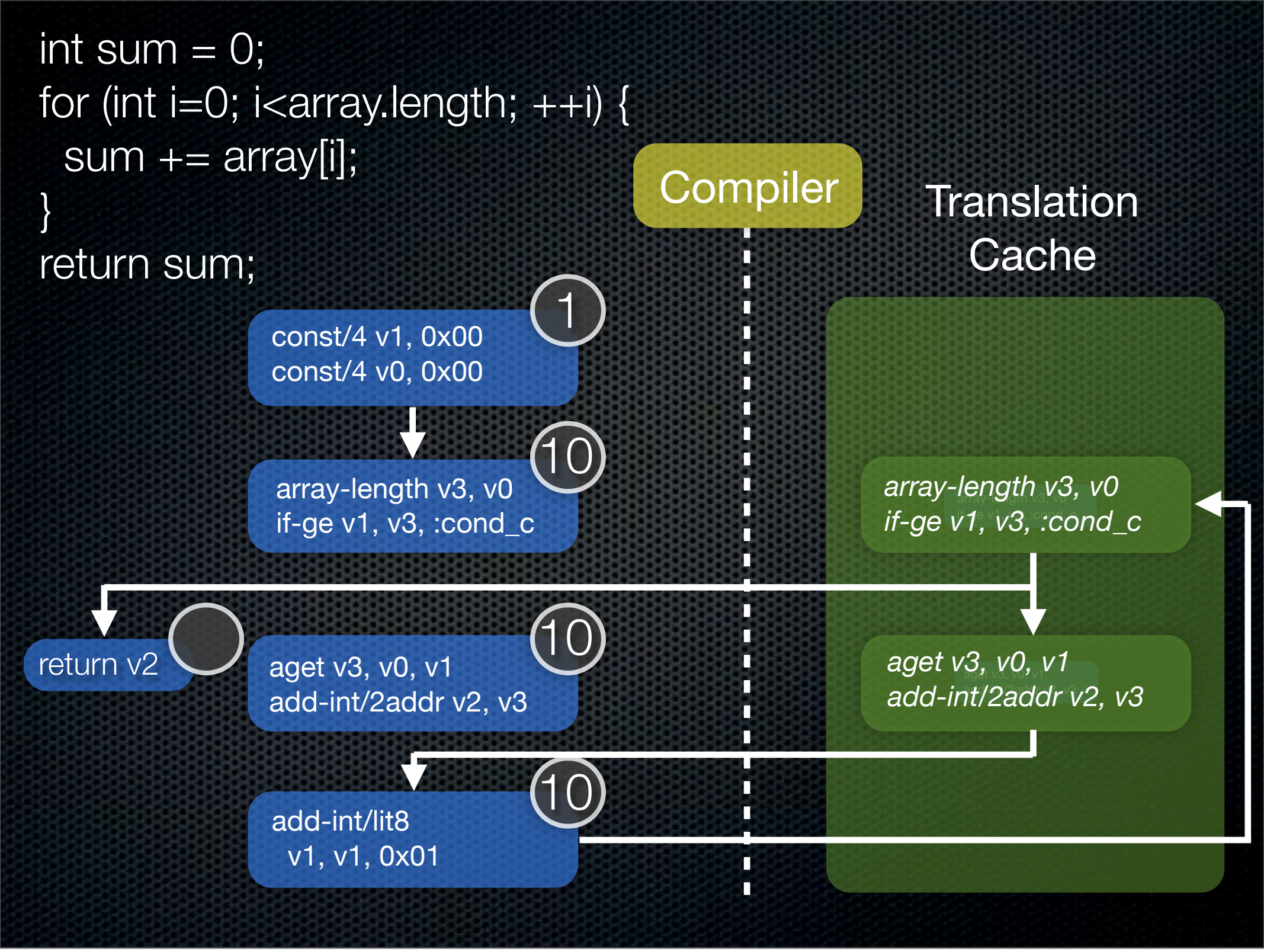
Keep count for each trace head (i.e. first line of block)

If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

Keep count for each trace head (i.e. first line of block)

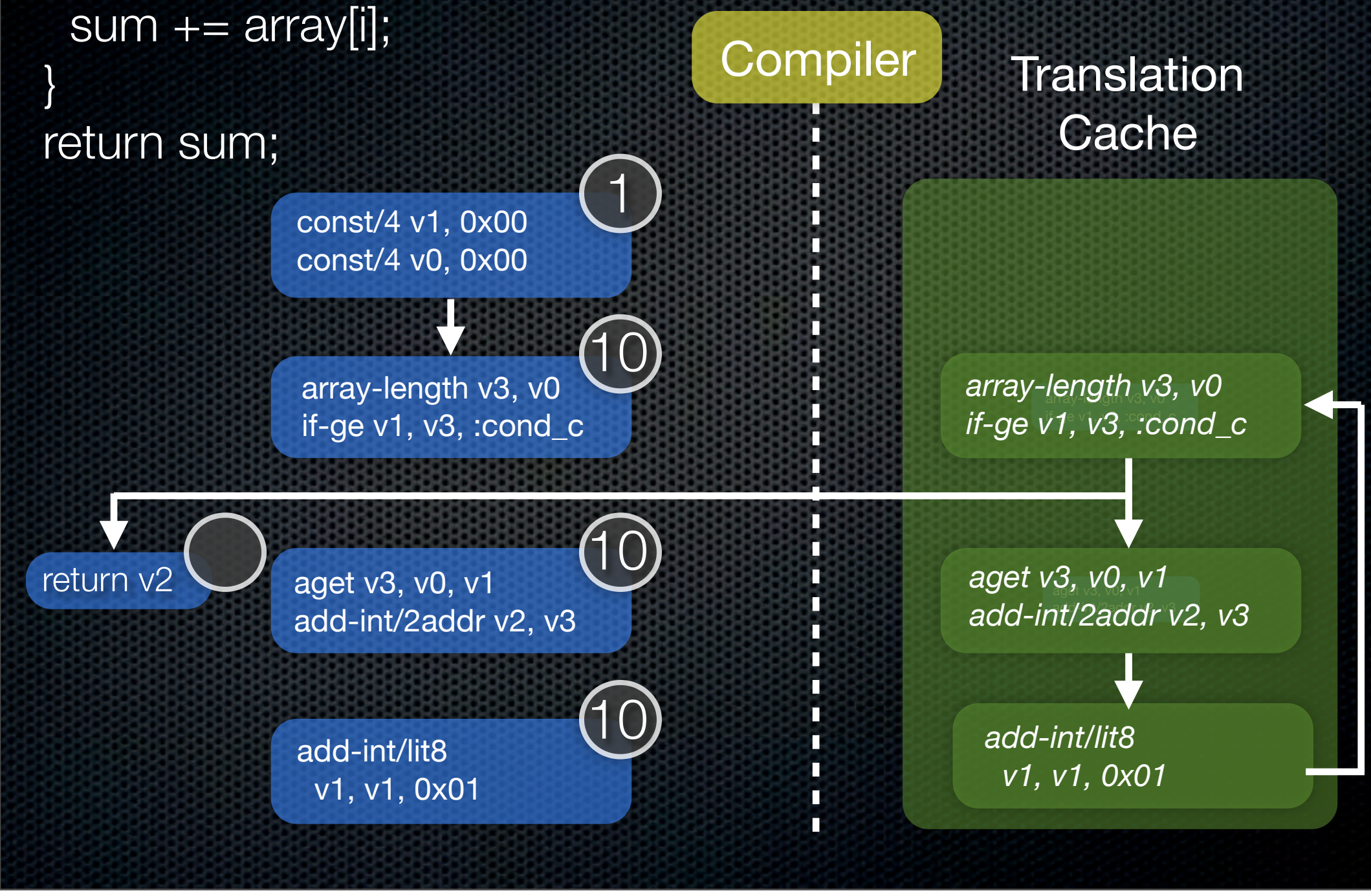
If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted

```
int sum = 0;
for (int i=0; i<array.length; ++i) {
  sum += array[i];
}
return sum;
```



http://www.youtube.com/watch?v=Ls0tM-c4Vfo

Keep count for each trace head (i.e. first line of block)

If a block reaches the threshold, hand over for translation

Once compilation is complete, the interpreter starts routing to the generated machine code

However, not all blocks may be compiled, so control has to return to the interpreter

Eventually, all hot blocks are compiled to machine code -- only the infrequently executed blocks, the initialization and return continue to be interpreted

Performance Recommendations?

Use Enhanced For Loop
Avoid Creating Objects
Use Native Methods (Judiciously)
Prefer Static Over Virtual
Prefer Virtual Over Interface
Avoid Internal Getters / Setters
Declare Constants Final
Avoid Enums
Use Package Scope with Inner Classes
Avoid Floats

http://developer.android.com/guide/practices/design/performance.html
There are a lot of performance claims out there, but performance recommendations typically only last for one or two VM revs.
Earlier Android advice might be just as wrong as early JVM advice is today, let's see...

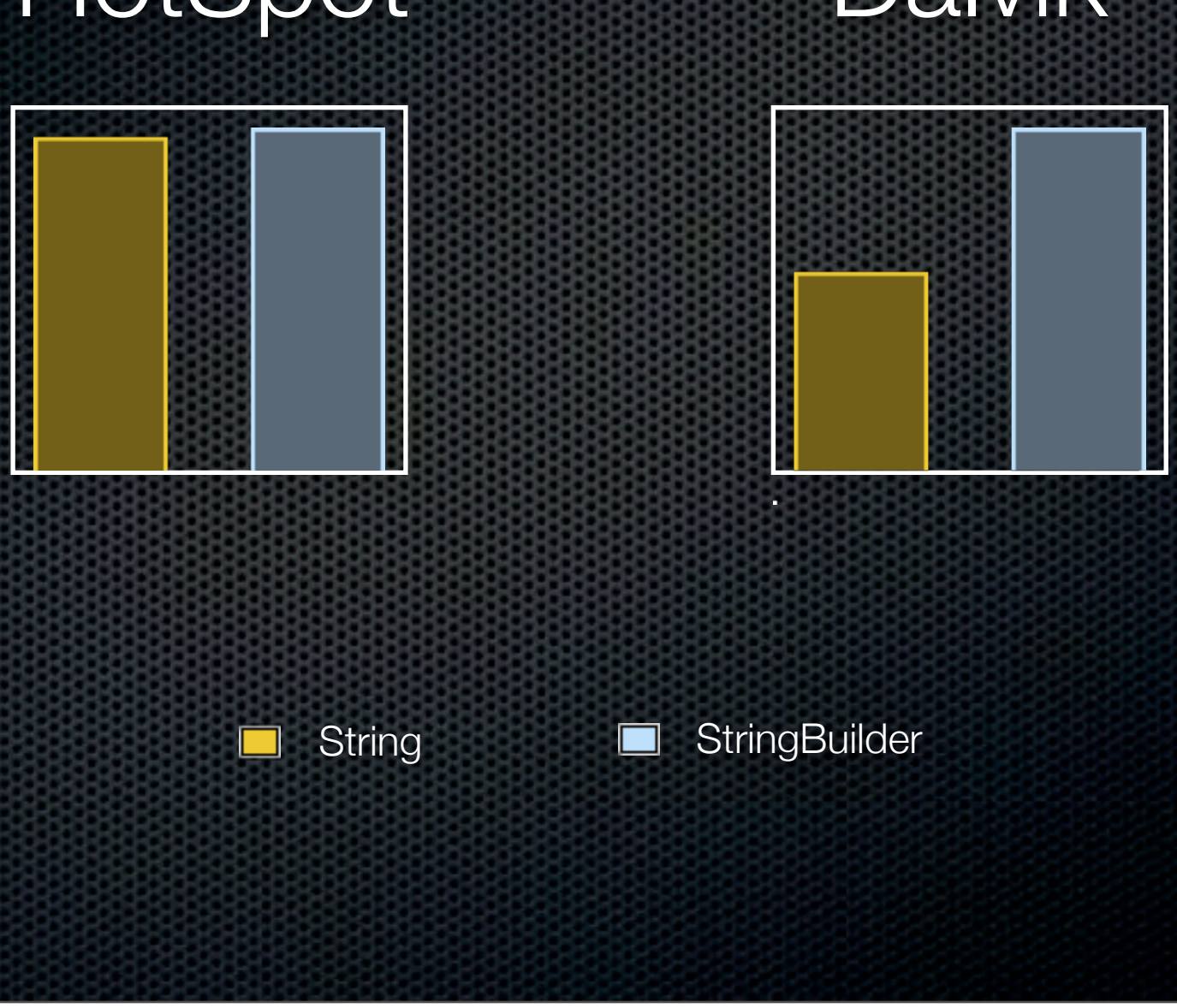
Demo

Caliper + vogar

Going to use one of my favorite tools: Caliper for microbenchmarking
Plus vogar, which makes it easy to run all types of tests: VM ref tests, JUnit tests, and Caliper benchmarks on the Android devices or the emulator

One note of caution these results are from a Xoom which may have a better JIT than an Android phone
http://code.google.com/p/caliper/
http://code.google.com/p/vogar/

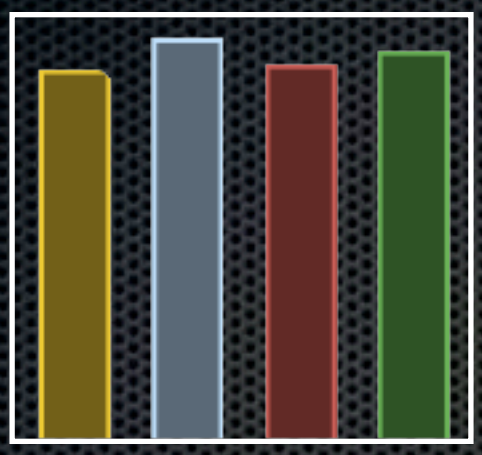
String Length



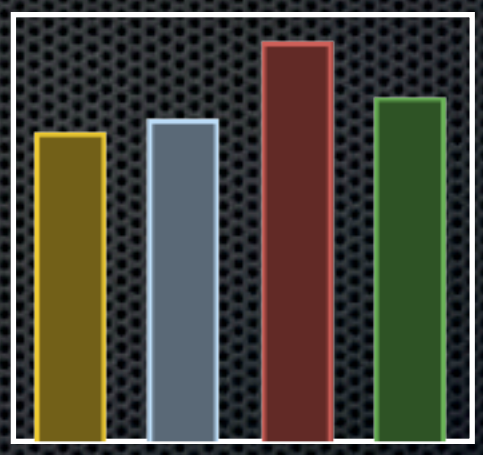
First, I wanted to see if the String.length claims were true, so I compared String.length performance to StringBuilder.length performance
On HotSpot, we can they come out the same, but, on Dalvik, the String.length takes half the time of StringBuilder.

Invocations

HotSpot



Dalvik

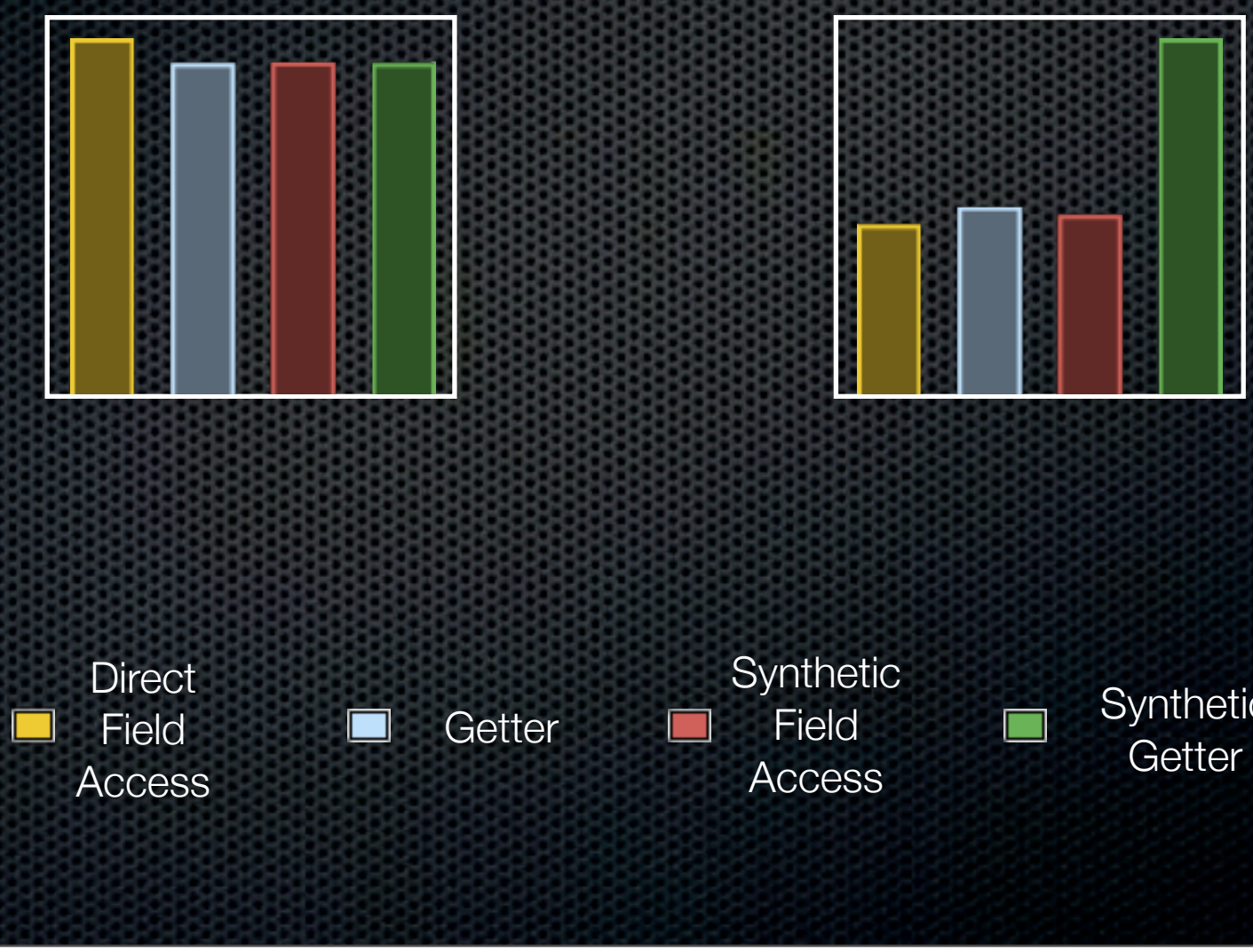


Static Private Virtual Interface

The results here are a bit inconclusive, but small methods almost all get inlined in HotSpot
Even polymorphic calls

In Dalvik, static and private appear to be slightly faster, but it does not make sense that invoke-virtual is slower than invoke-interface. Although, preferring virtual over interface seems questionable.

Synthetic Access

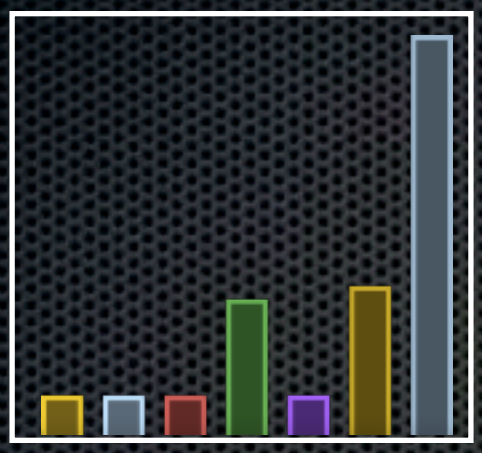


In HotSpot, synthetic access makes little difference (despite the warnings). Synthetic accessors are smaller than the 15-byte automatic inline threshold, so they make little difference.

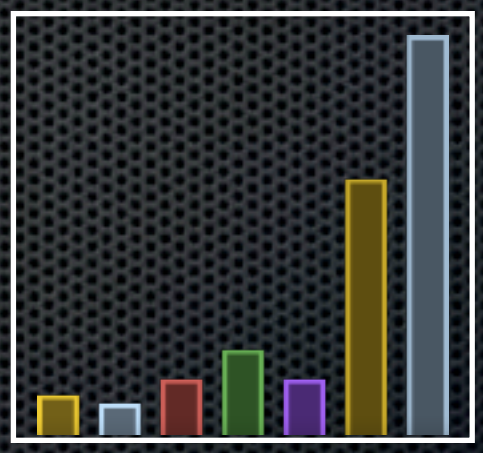
In Dalvik, it is largely the same. Getters get inlined. A synthetic field accessor is just a getter, so it gets inlined. But, a synthetic method accessor incurs a significant penalty.

Loops

HotSpot



Dalvik



Primitive Backwards

Primitive Enhanced For

Primitive Tranditional For

Boxed Enhanced For

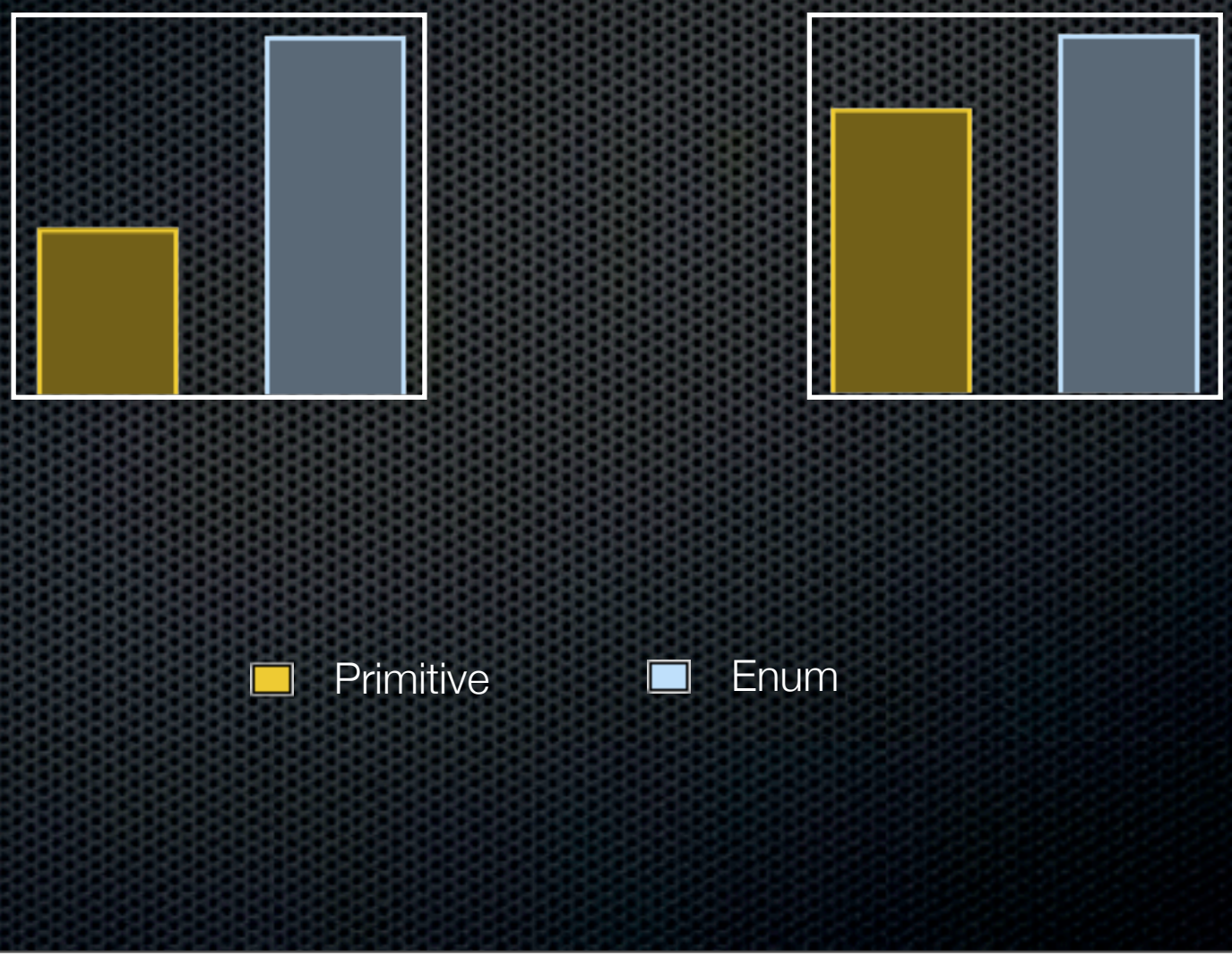
Boxed Traditional For

ArrayList Thread Unsafe

ArrayList Iterator

Loops...
Primitive arrays with the new or old for loop perform equally well in both
HotSpot new for reading may be an anomaly.
Definitely, take a hit in both from using a List.
Especially, when using an Iterator - i.e. new for.

Enums



Somewhat surprisingly at least in relative terms, Dalvik seems to out perform HotSpot when working when enums