

# Application parallelization for multi-core Android devices

Jos van Eijndhoven  
jos@vectorfabrics.com

**Bits&Chips 2012 Embedded Systems**  
Nov. 8, 2012, 's-Hertogenbosch



# Multi-core ARM and Android conquer the world



Google Nexus 10  
2-core Samsung A15



Asus Transformer Prime  
“4”-core Nvidia Tegra3



HTC J Butterfly  
4-core Qualcomm



Samsung Galaxy SIII  
4-core Samsung



Sony Xperia P  
2-core ST-Ericsson



Huawei Honor2  
4-core Huawei

# Multi-core usage in Mobile

- 2 core processors:  
Assume the OS has **multiple processes** and/or kernel threads to occupy the two cores. **Easy!**
- 4 core processors (and beyond):  
Requires **multi-threaded applications** **Hard!**
  - To obtain sufficient concurrent workload
  - To obtain top user experience

*Who makes such applications??*

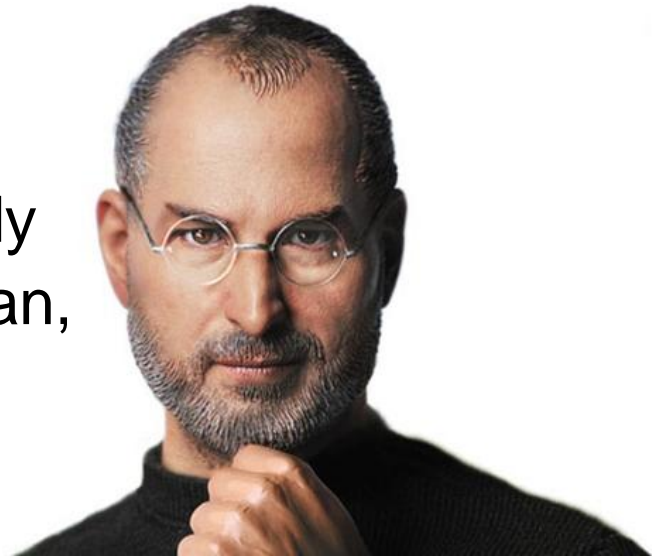
# Creating parallel programs is hard...

Herb Sutter, chair of the ISO C++ standards committee,  
Microsoft:

“Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren’t possible, and discovers that they didn’t actually understand it yet after all”

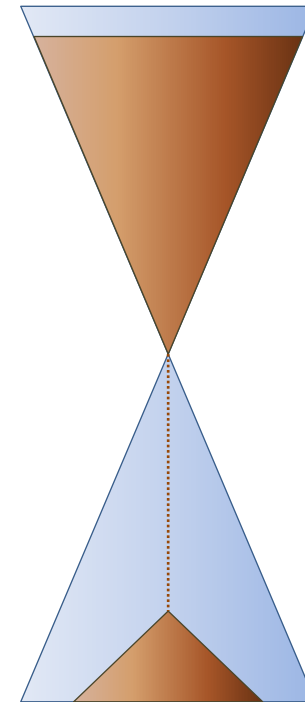
Steve Jobs, Apple:

“The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two yeah; four not really; eight, forget it.”



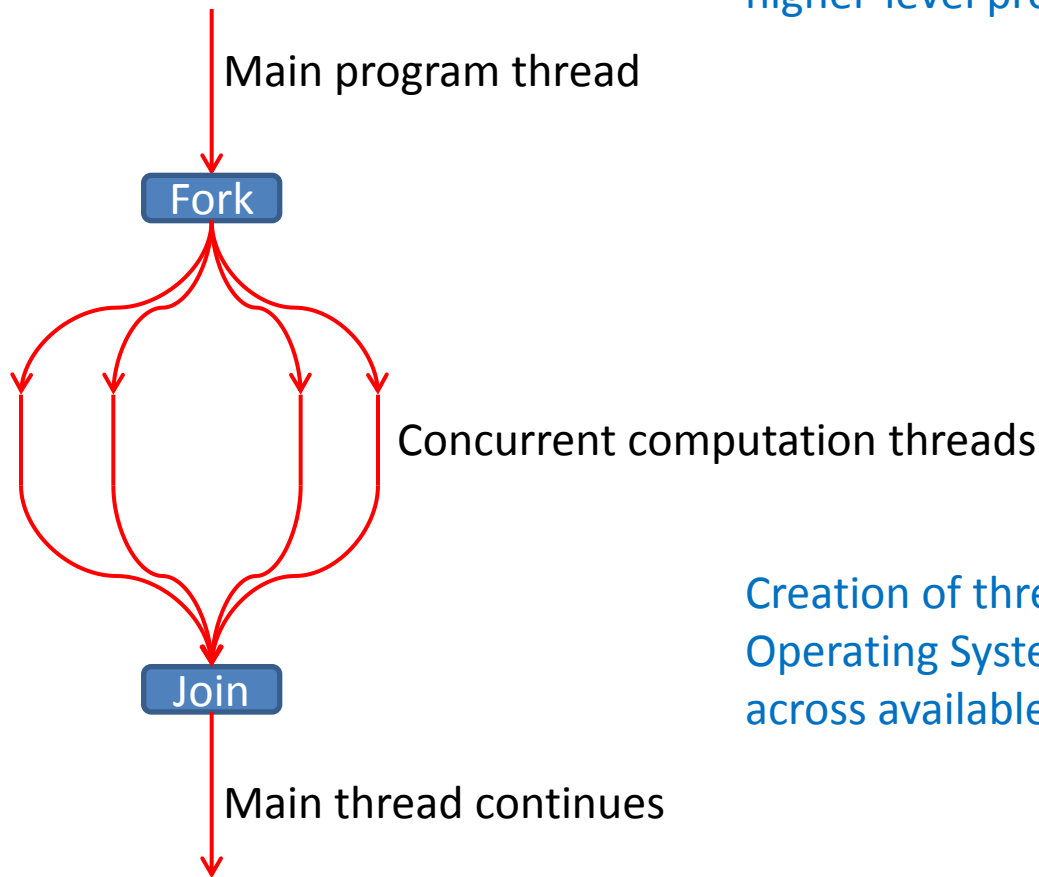
# Presentation index

- Introduction
- Multi-threaded concurrency:  
Data- versus Task-partitioning
- Parallelization with dependencies:  
Reduction expressions or Streaming
- Multi-threading: difficult...
- Android: help from Parelion and Perf
- Conclusion



# Creating multi-threaded concurrency

Basic fork-join pattern, created through different higher-level programming constructs



Creation of threads is application responsibility. Operating System handles run-time scheduling across available processors!

# Parallelization – two partitioning options

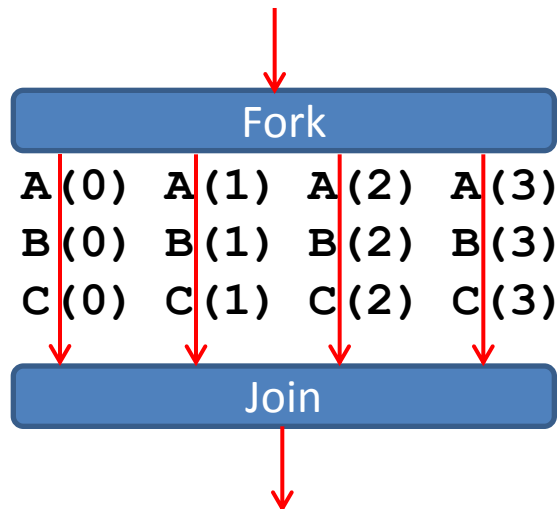
## Source code:

```
for (i=0; i<4; i++) {  
    A(i);  
    B(i);  
    C(i);  
}
```

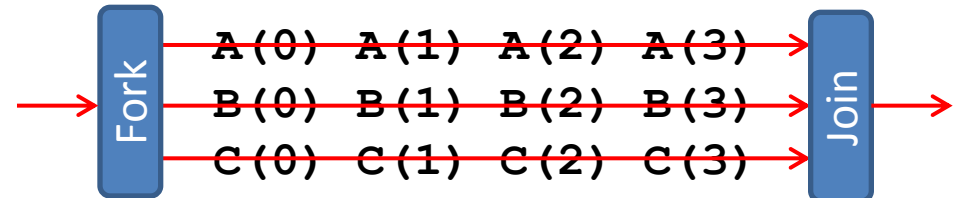
## Sequential execution order:

A(0) A(1) A(2) A(3)  
B(0) B(1) B(2) B(3)  
C(0) C(1) C(2) C(3)

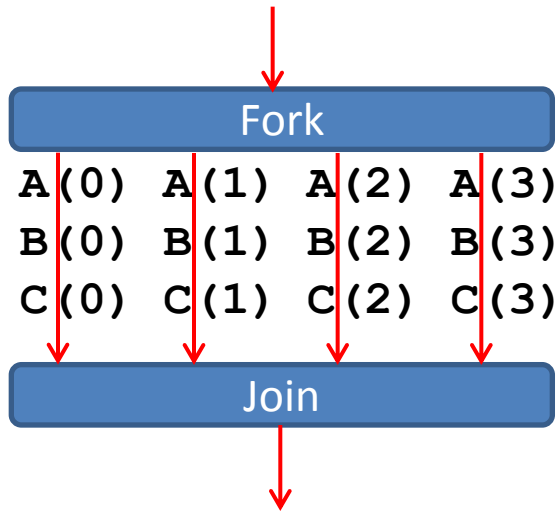
## Data partitioning:



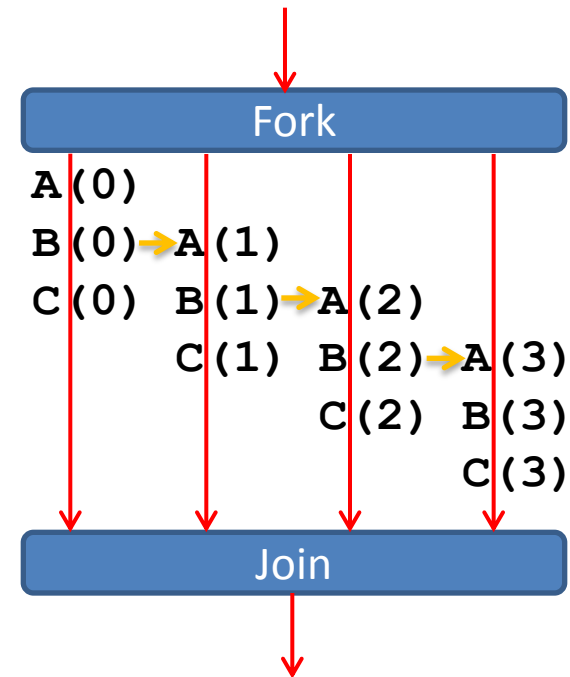
## Task partitioning:



# Issue: Data dependencies



Maybe,  $B(i)$   
produces a value  
that is used by  
 $A(i+1)$ ...



Adjust program source for parallelization:

- When feasible, remove inter-thread data dependencies
- Implement required data synchronization



# Example Data dependencies

## *Variable assigned in loop body, used in later iteration*

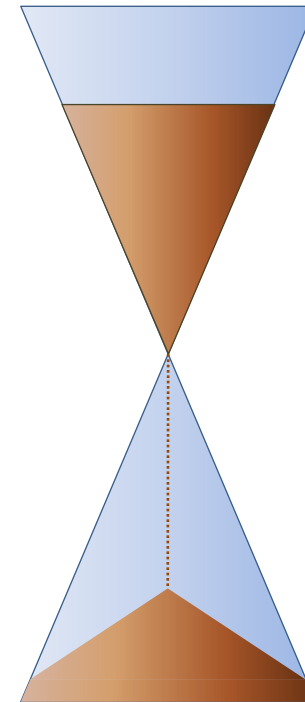
```
// search linked-list for matching items
// save matches in 'found' array of pointers
for (p = head, n_found = 0; p; p = p->next)
    if (match_criterion(p))
        found[n_found++] = p;
```

Cannot (easily/trivially) spawn data-parallel tasks!

- No direct parallel access to list members **\*p**
- No direct way to assign index to matched item **n\_found**
- Maybe more problems hidden in **match\_criterion()**

# Presentation index

- Introduction
- Multi-threaded concurrency:  
Data- versus Task-partitioning
- Parallelization with dependencies:  
Reduction expressions or Streaming
- Multi-threading: difficult...
- Android: help from Parelion and Perf
- Conclusion



# Can do: reduction data dependencies

- Reduction expressions: accumulate results of loop bodies with commutative operations
- Freedom of re-ordering allows to break sequential constraints

```
// conditionally accumulate results
```

```
int acc = 0;
for (i=0; i<N; i++)
{
    int result = some_work(i);
    if (some_condition(i))
        acc += result;
}
```

```
...use of acc ...
```

- Commutative operations are basic math like +, \*, &&, &, ||, but also more complex operations like 'add item to set'.
- Three(?) different methods to handle these ...

# Three methods for reduction dependencies

- Create thread-local copies of the accumulator. Accumulate over local copy in each thread. Merge the partial accumulators after thread-join. Eg. created automatically by:

```
#pragma omp parallel for reduction(...)
```

- Maintain single accumulator, synchronize updates through atomic operations. Eg. in C11 or C++11:

```
atomic_add_fetch( &acc, result);  
std::atomic<int> acc;  
acc += result;
```

- Maintain single accumulator, synchronize updates through protection by acquiring and releasing semaphores. Eg. Used by Intel “Threaded Building Blocks” (C++):

```
concurrent_unordered_set<...> s;  
s.insert(...);
```

# PAREON: Schedule data dependencies

The screenshot displays the PAREON software interface, which is used for analyzing and optimizing code execution. The interface is divided into several panels:

- Partitioning candidates - Loop\_38:** This panel shows CPU data partitioning options. The number of threads is set to 4. Performance metrics include a global speedup of 2.3, a global overhead of 6%, and a thread creation delay of 420 us. A table lists the partitioning candidates:

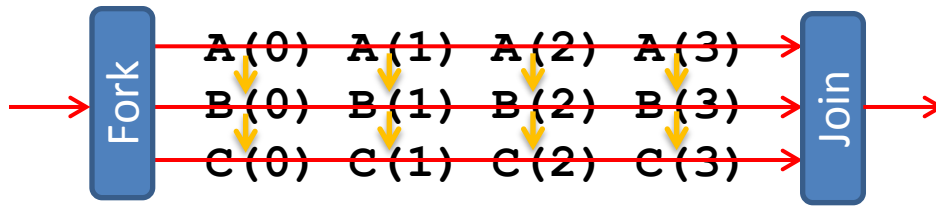
Invocation	Speedup	Overhead	Streams
Loop_38	3.9	1 %	1

- Properties / My changes:** This panel shows the configuration for the selected loop (Loop\_38). Key properties include an iteration count of 150, a computation time of 85.3 us (92.7%), and a memory penalty of 6.8 us (7.3%).
- 2D-Profile / Schedule:** This panel provides a visual representation of the schedule data dependencies. It shows a "Schedule overview" with a progress bar indicating 99% execution. Below this, a "Schedule execution" diagram illustrates the flow of data dependencies between iterations, showing a zig-zag pattern of dependencies across iterations #68 to #83.

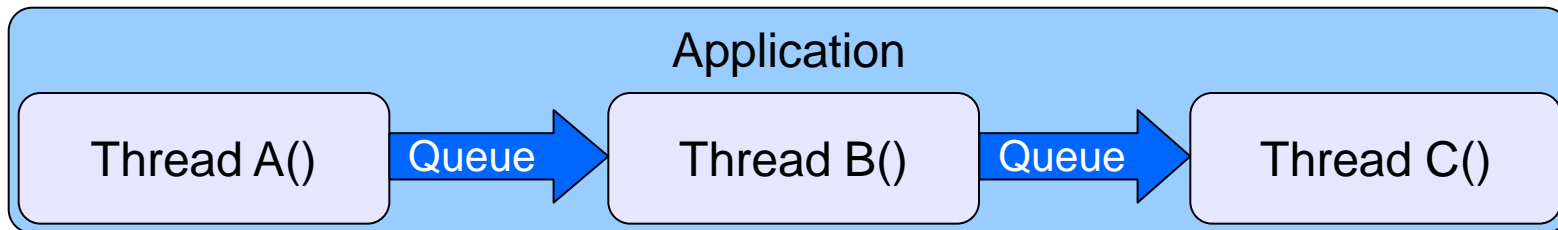
A blue callout box at the bottom right of the 2D-Profile panel contains the text: "Note: this is a *preview* on a potential parallelization".

# Pipelining: Data deps and task partitioning

## Task partitioning with inter-thread dependencies:



## Producer-Consumer pattern:

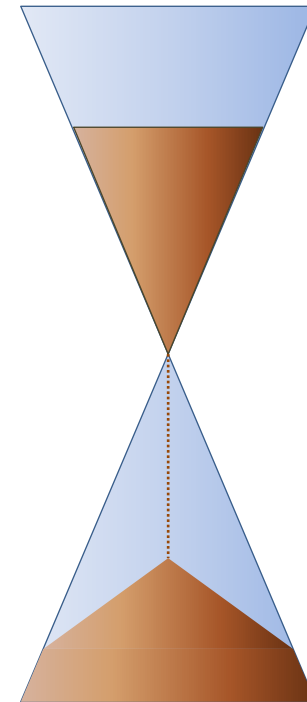


Queue implementation solves dependencies:

- **Solve Data dependencies:** Consumer thread waits for available data (stalls until queue is non-empty)
- **Solve Anti dependencies:** Producer thread creates next item in next memory location (prevents overwriting previous value)

# Presentation index

- Introduction
- Multi-threaded concurrency:  
Data- versus Task-partitioning
- Parallelization with dependencies:  
Reduction expressions or Streaming
- **Multi-threading: difficult...**
- **Android: help from Pareon and Perf**
- Conclusion



# Concurrent C/C++ programming: Pitfalls

## Risc introduction of functional errors:

- Overlooking use of shared/global variables (deep down inside called functions, or inside 3rd party library)
- Overlooking exceptions that are raised and caught outside studied scope
- Incorrect use of semaphores: flawed protection, deadlocks

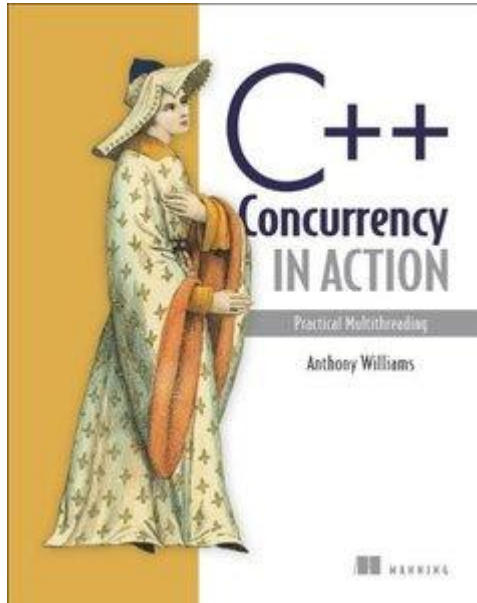
## Unexpected performance issues:

- Underestimation of time spent in added multi-threading or synchronization code and libraries
- Underestimation of other penalties in OS and HW (inter-core cache penalties, context switches, clock-frequency reductions)

## Parallel programming remains hard!



# Concurrent programming remains hard



- C++11 standardizes valuable primitives
- Provides good insight in C++ concurrency
- Warns for many subtle problems

# Development of parallel code

## Guidelines:

- Base upon a sequential program:  
functional and performance reference
- Apply higher-level parallelization patterns and primitives:  
clear semantics, re-use code, reduce risk
- Use tooling for analysis and verification
  - Prevent introduction of hard-to-find bugs
  - Prevent recoding effort that does not perform

## Managable development process!

# PAREON tool workflow

1

Developer builds C/C++ application with Pareon's compiler  
Compiler creates instrumented code



2

Execute application with input data set  
Pareon captures trace with call stack and ld/st memory traffic  
Pareon analyzes data dependency patterns

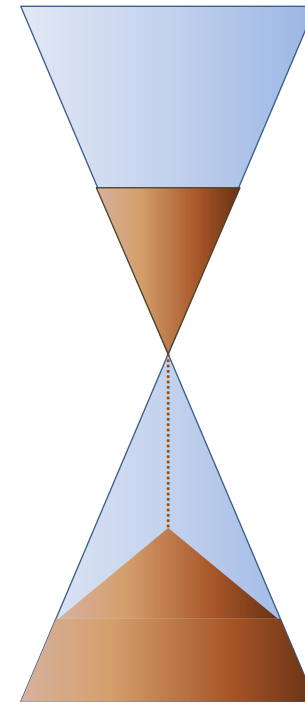


3

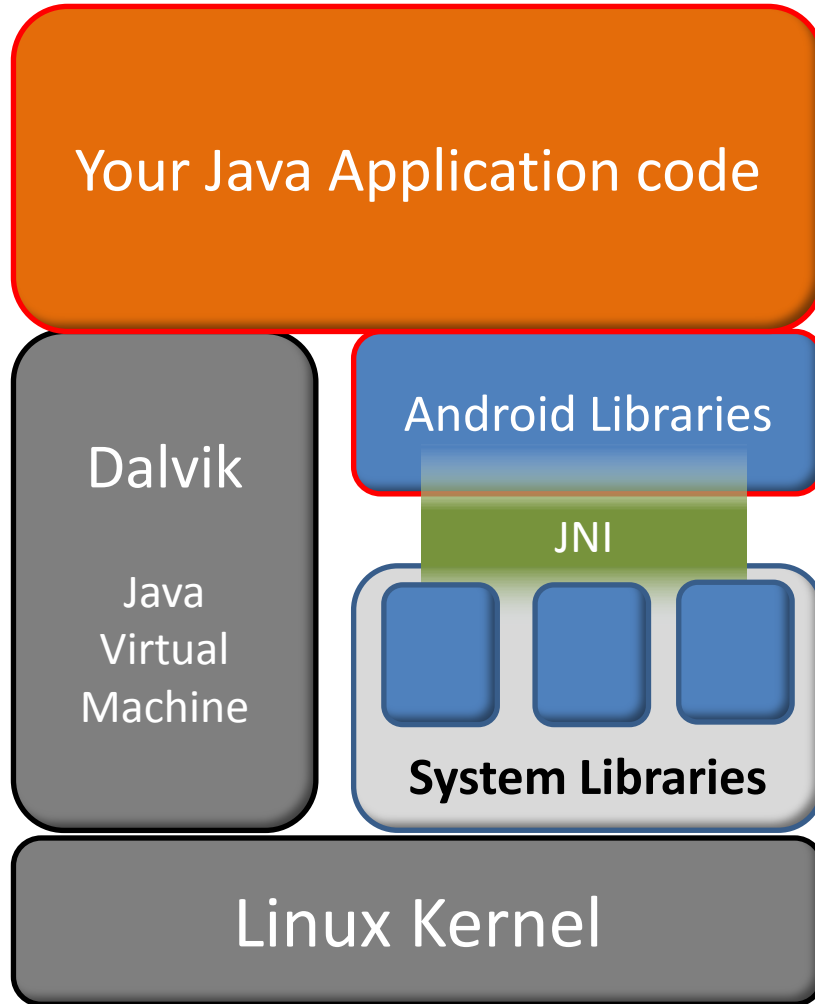
Developer browses application model  
Selects loops for parallelization from performance estimate  
Performs source-code transformation

# Presentation index

- Introduction
- Multi-threaded concurrency:  
Data- versus Task-partitioning
- Parallelization with dependencies:  
Reduction expressions or Streaming
- Multi-threading: difficult...
- **Android: help from Pareon and Perf**
- **Conclusion**



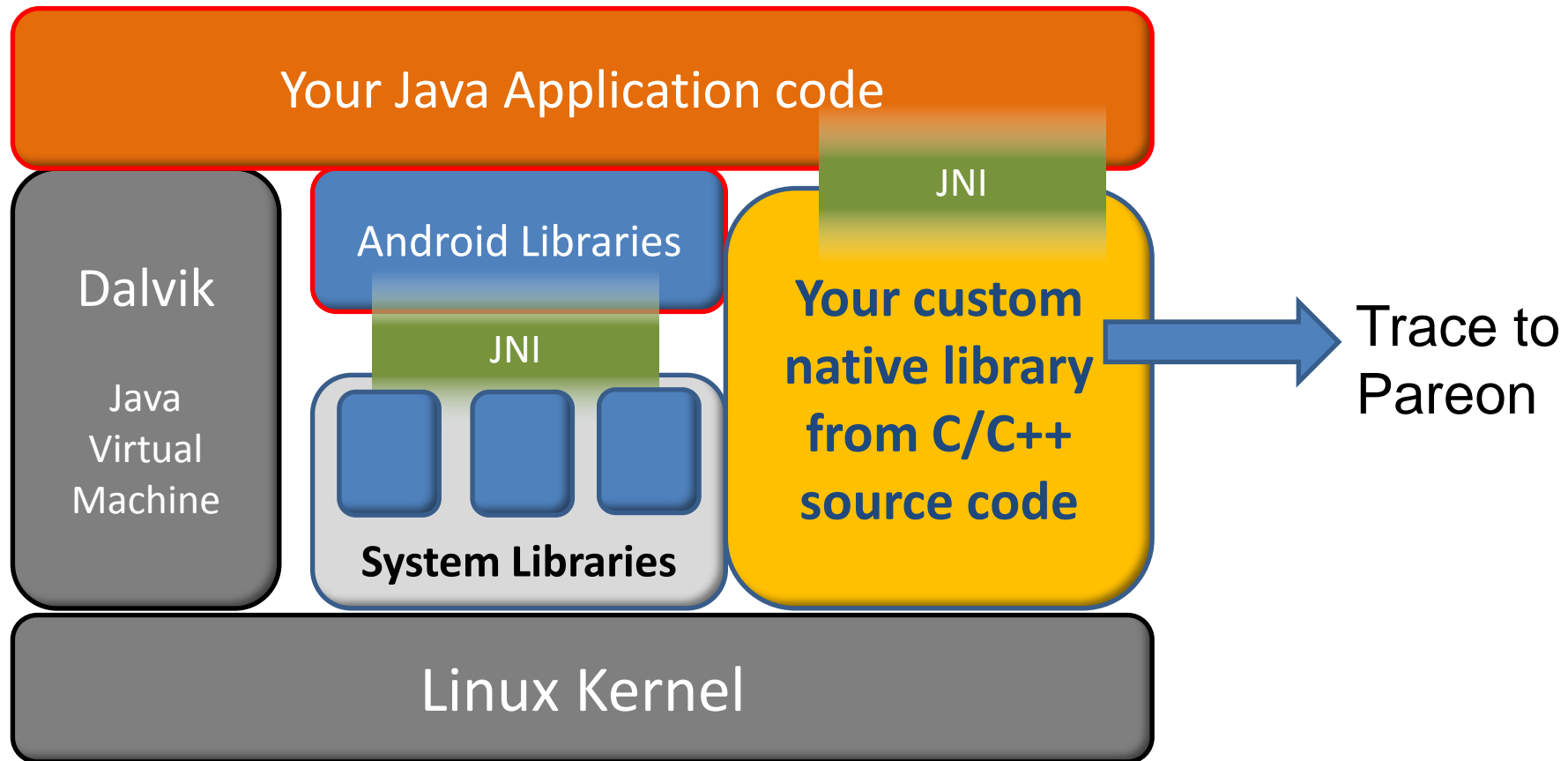
# Android Application 1: Plain, just Java



Many apps have no critical CPU load  
For now, no Java support in Pareon

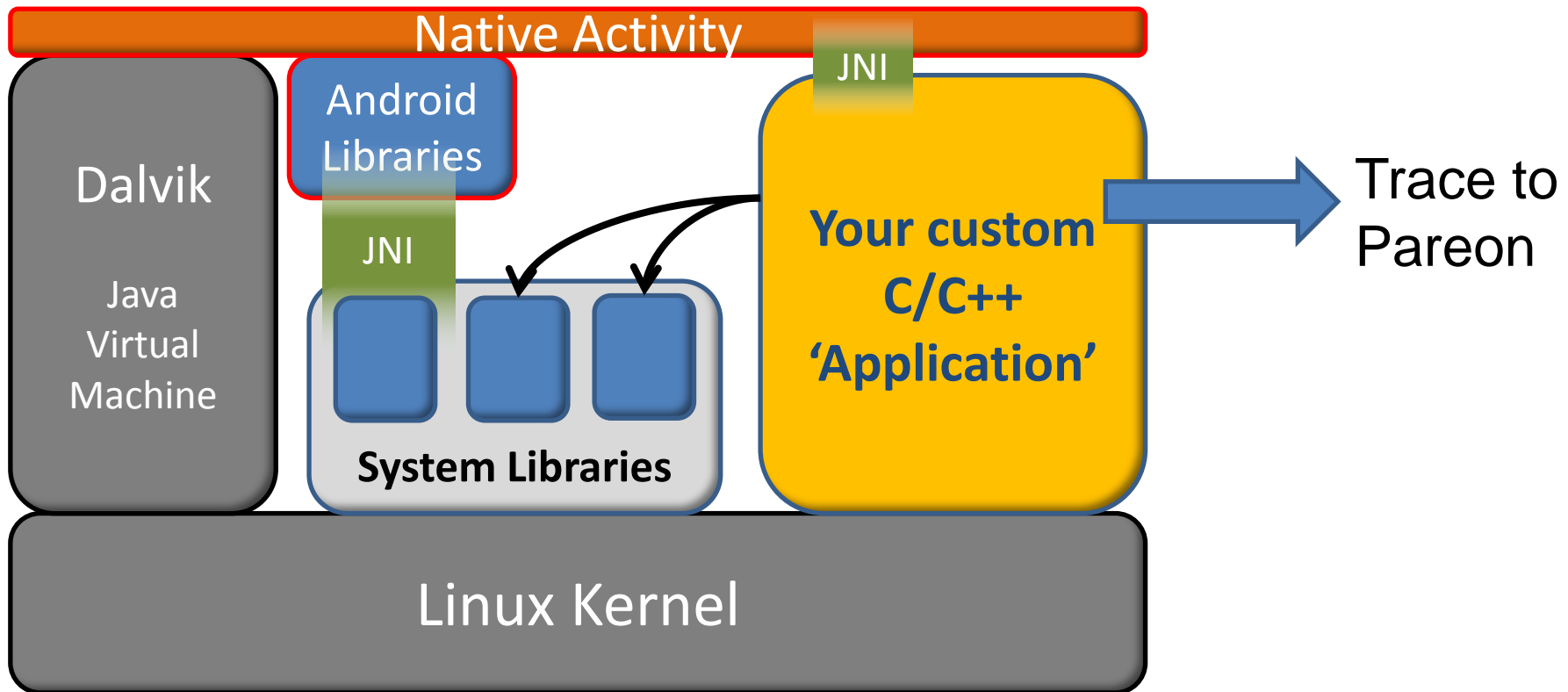
# Android Application 2: with native libraries

Apps can include “native” binary code for best performance

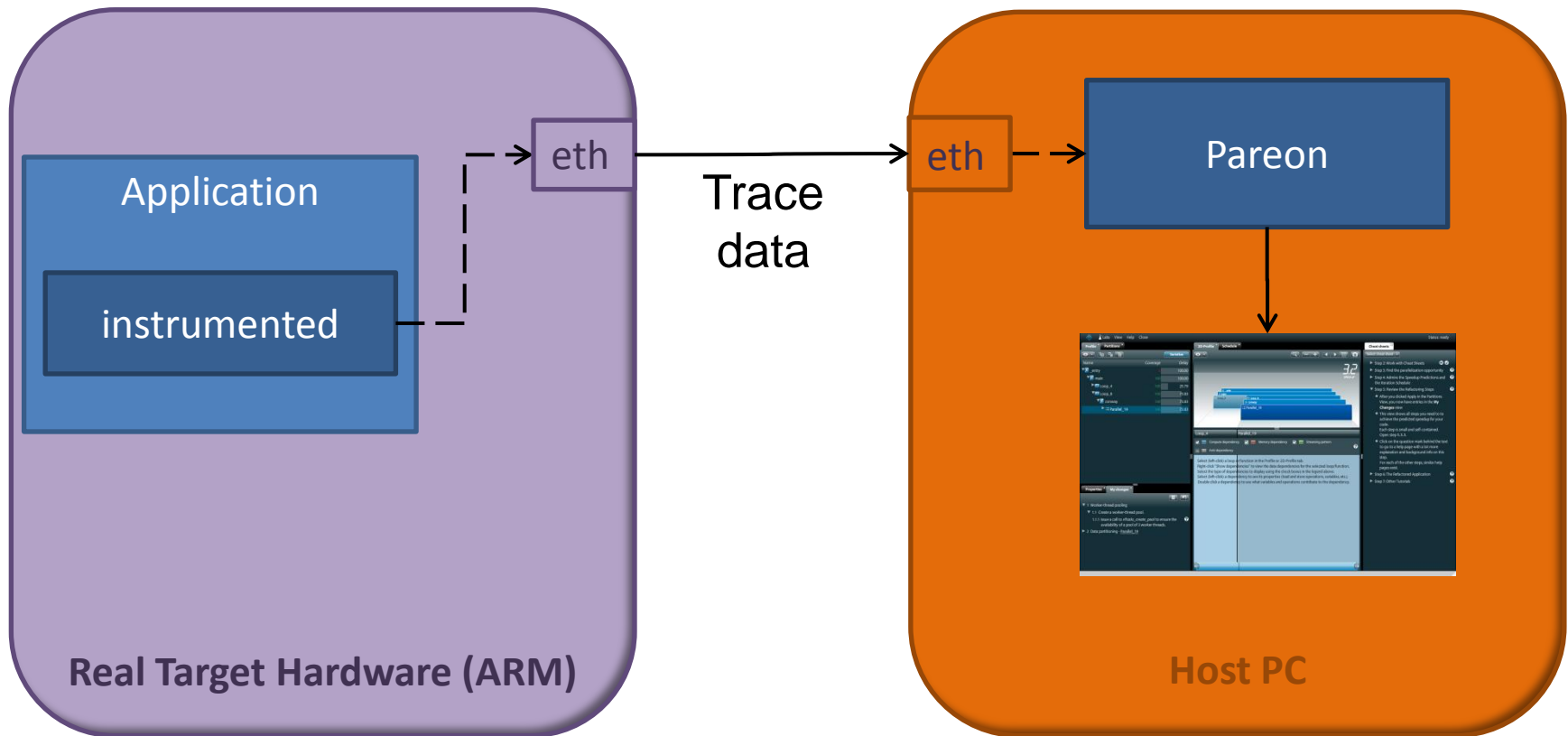


# Android Application 3: NativeActivity

“Native activities” are created without Java source code

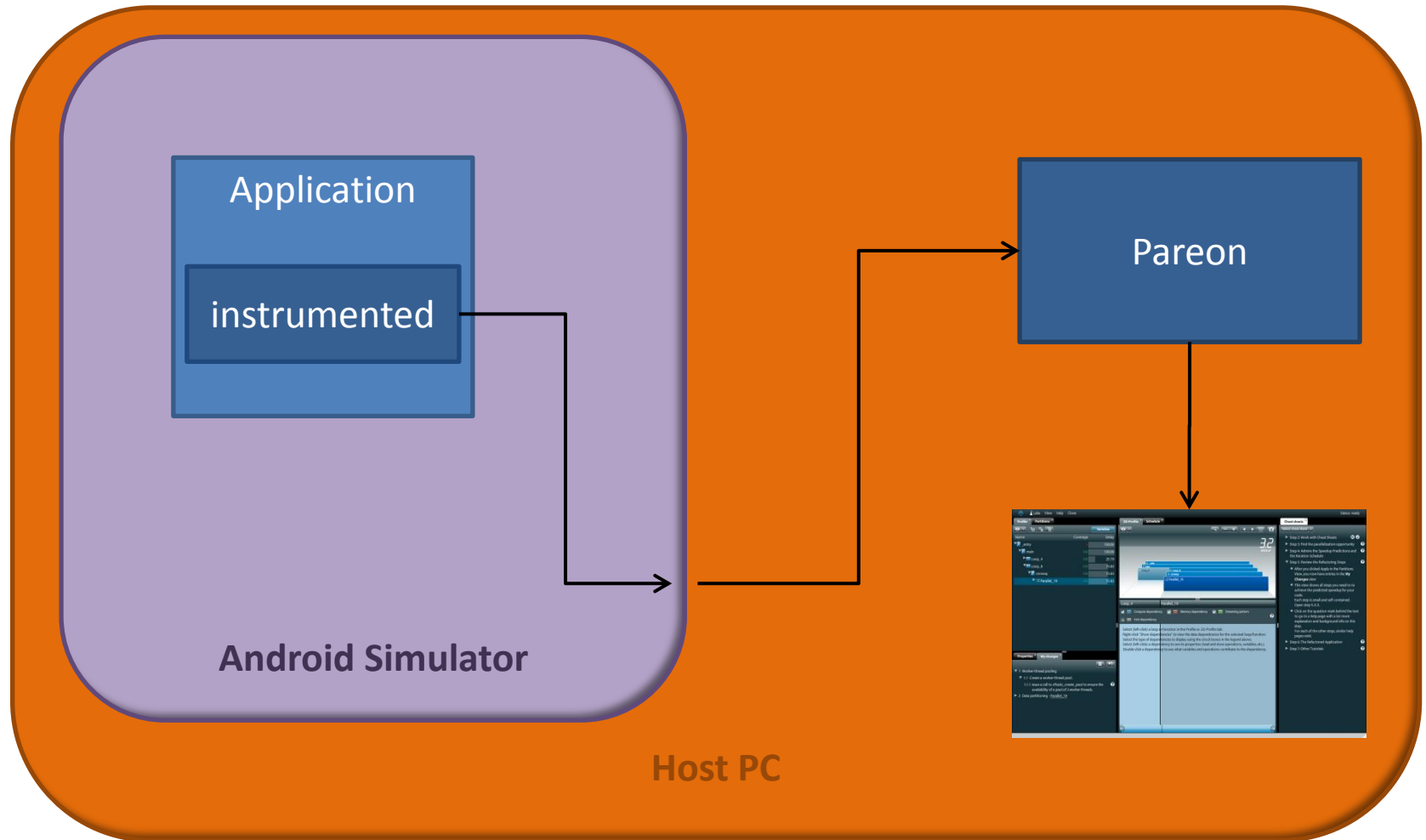


# Application Analysis on Android target





# System Setup using Android Simulator



# NDK plasma demo app analyzed on Android



# Finding data parallelism on Android

The screenshot displays the VectorFabrics Labs software interface, which is used for analyzing and optimizing code for data parallelism. The interface is divided into several panels:

- Profile / Partitions:** Shows partitioning candidates for Loop\_313. The number of threads is set to 4. The global speedup is 2.4, and the global overhead is 1%. A table lists the candidates with their speedup and overhead values.
- Properties / My changes:** Shows the properties of the selected partitioning candidate, including the source code location and the number of loop-carried transfers.
- 2D-Profile / Schedule / plasma.c:** Shows a 2D profile of the code, highlighting the Loop\_313. The profile shows the execution of the loop across multiple threads, with a speedup of 1.0.

**Partitioning candidates - Loop\_313**

▼ CPU data partitioning - vfTasks

Number of threads: 4 **Apply**

Global speedup: 2.4    Extra worker threads: 3  
Global overhead: 1%    Thread creation delay: 420 us

<input checked="" type="checkbox"/>	Invocation	Speedup	Overhead	Streams
<input checked="" type="checkbox"/>	Loop_120	4.0	1%	0
<input checked="" type="checkbox"/>	Loop_189	4.0	1%	0
<input checked="" type="checkbox"/>	Loop_313	3.9	1%	0

**Properties / My changes**

Property	Value
▼ Compute dependency 313.46	
Ignore with data partitioning	induction expression
▼ Source	
Operation (+)	Loop_313 (fill_plasma)
Location	plasma.c:211
▶ Destinations	
#loop-carried transfers	68.2 Ki transfers/s
Loop carried	yes

**2D-Profile / Schedule / plasma.c**

Loop\_313 total loop carried transfer rate: 136 Ki transfers/s  
0 streaming pattern clusters (0.0 transfers/s); 0 data dependency clusters (0.0 transfers/s);  
2 compute dependencies (136 Ki transfers/s); 0 anti- and output dependency clusters

# Finding data parallelism on Android

The screenshot displays the Vectorworks IDE interface, divided into several panels. The top-left panel, titled 'Partitions', shows 'Partitioning candidates - Loop\_313' with a table of CPU data partitioning options. The 'Number of threads' is set to 4. Below this, a table lists candidates for Loop\_120, Loop\_189, and Loop\_313, with their respective speedups and overheads. The bottom-left panel, 'Properties', shows details for 'Loop\_313', including its source location in 'plasma.c:211' and its transfer rate of 68.2 Ki transfers/s. The right-hand side of the interface features a '2D-Profile' window for 'plasma.c', which visualizes the execution flow of various loops. A red circle highlights the 'Loop\_313' node in the profile, and another red circle highlights the 'Loop\_313' entry in the 'Partitions' table. A third red circle highlights the 'Loop\_313' entry in the 'Properties' table. The bottom-right panel shows a detailed view of the 'Loop\_313' node, including its dependencies and transfer rates.

Profile x Partitions x

Partitioning candidates - Loop\_313

▼ CPU data partitioning - vfTasks

Number of threads  **Apply**

Global speedup: 2.4    Extra worker threads: 3  
Global overhead: 1%    Thread creation delay: 420 us

<input checked="" type="checkbox"/>	Invocation	Speedup	Overhead	Streams
<input checked="" type="checkbox"/>	Loop_120	4.0	1%	0
<input checked="" type="checkbox"/>	Loop_189	4.0	1%	0
<input checked="" type="checkbox"/>	Loop_313	3.9	1%	0

Properties x My changes x

Property	Value
▼ Compute dependency 313.46	
Ignore with data partitioning	induction expression
▼ Source	
Operation (+)	Loop_313 (fill_plasma)
Location	plasma.c:211
▶ Destinations	
#loop-carried transfers	68.2 Ki transfers/s
Loop carried	yes

2D-Profile x Schedule x plasma.c x

1.0 SPEED-UP

Loop\_313

Loop\_313 total loop carried transfer rate: 136 Ki transfers/s  
0 streaming pattern clusters (0.0 transfers/s); 0 data dependency clusters (0.0 transfers/s);  
2 compute dependencies (136 Ki transfers/s); 0 anti- and output dependency clusters



# Finding data parallelism on Android

The screenshot displays the VectorFabrics software interface, divided into several panels:

- Partitions Panel:** Shows partitioning candidates for Loop\_313. The "Number of threads" is set to 4 (circled in red). Other metrics include Global speedup: 2.4, Extra worker threads: 3, Global overhead: 1%, and Thread creation delay: 420 us. A table lists candidates with their speedup and overhead.
- Properties Panel:** Shows details for Loop\_313, including its source location (plasma.c:211) and transfer rates.
- 2D-Profile Panel:** Displays a 3D bar chart of execution times for various loops, with Loop\_313 highlighted. A speedup indicator shows 1.0.
- Legend Panel:** Defines dependency types: Compute dependency (blue), Memory dependency (orange), Streaming pattern (green), and Anti-dependency (grey).
- Summary Panel:** Provides a detailed breakdown of Loop\_313's performance, including a total loop transfer rate of 136 Ki transfers/s and a list of dependency clusters.

Candidate	Speedup	Overhead	Streams
Loop_120	4.0	1 %	0
Loop_189	4.0	1 %	0
Loop_313	3.9	1 %	0

Property	Value
Compute dependency	313.46
Ignore with data partitioning	induction expression
Source	
Operation (+)	Loop_313 (fill_plasma)
Location	plasma.c:211
Destinations	
#loop-carried transfers	68.2 Ki transfers/s
Loop carried	yes

Loop\_313 total loop carried transfer rate: 136 Ki transfers/s  
0 streaming pattern clusters (0.0 transfers/s); 0 data dependency clusters (0.0 transfers/s);  
2 compute dependencies (136 Ki transfers/s); 0 anti- and output dependency clusters

# Not parallelized: JNI call to render frame

**Profile** **Partitions** **2D-Profile** **Schedule**

Name	Cover...	Delay
ANativeWindow_lock		6.05
__android_log_print		0.00
stats_startFrame	100	0.07
clock_gettime		0.23
fill_plasma	70	42.10
Parallel_370	64	42.10
ANativeWindow_unlockAndPost		26.17
stats_endFrame	93	1.61

**Properties** **My changes**

Property	Value
ANativeWindow_unlockAndf	
Intrinsic	ANativeWindow_unlockAndPost
Invocation time	
Estimated	1.9 ms
Constraint	<delay> ns
Invocation statistics	
Mapped to Instance	ARM-A9
Call location	plasma.c:407

2.4 SPEED-UP

ANati... P. en... AN... Parallel\_370 ANativeWindow\_unloc...

Compute dependency  Memory dependency  Streaming pattern

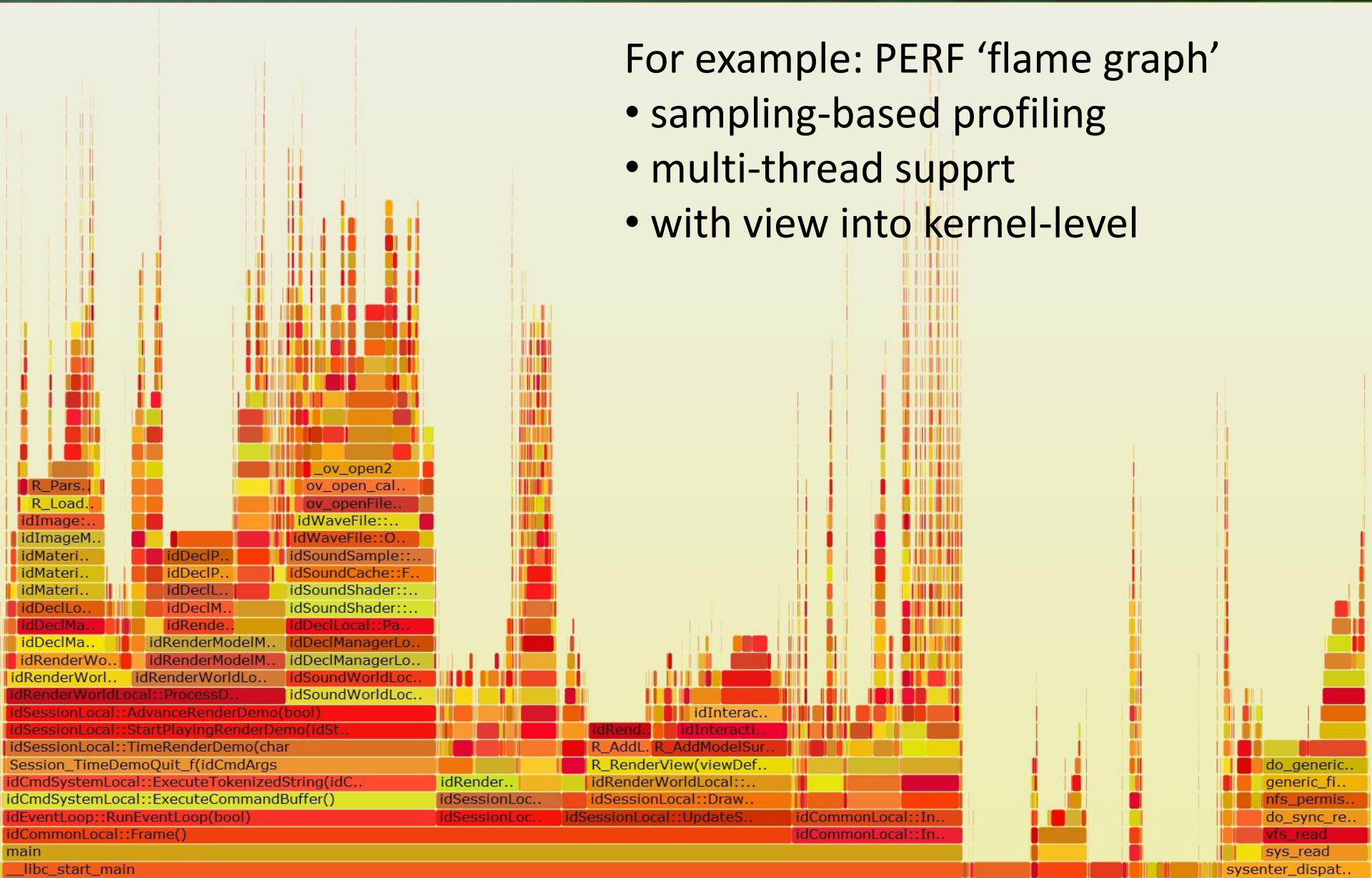
Anti-dependency

Select (left-click) a loop or function in the Profile or 2D-Profile tab.  
Right-click i.e. "Show internal dependencies" to view the internal data dependencies for the selected loop/function.  
Select the type of dependencies to display using the check boxes in the legend above.  
Select (left-click) a dependency to see its properties (load and store operations, variables, etc.).

# Performance Verification

For example: PERF 'flame graph'

- sampling-based profiling
- multi-thread support
- with view into kernel-level



# Conclusion

## Today's gap:

- Multi-core CPUs are everywhere,
- Yet multi-threaded programming remains hard:
  - Risk of creating hard-to-locate bugs regarding dynamic data races and semaphore issues
  - Obtained speedup is lower than expected
- A sequential functional reference implementation ...  
... helps to set a baseline for parallelization
- Android sets a new record in development complexity
- Proper tooling is needed to save on edit-verify development cycles



# Questions?

## Today's gap:

- Multi-core CPUs are everywhere
- Yet multi-threaded programming remains hard:
  - Risk of creating hard-to-locate bugs regarding dynamic data races and semaphore issues
  - Obtained speedup is low when expected
- A sequential functional reference implementation ...  
... helps to set a baseline for parallelization
- Android sets a new record in development complexity
- Proper tooling is needed to save on edit-verify development cycles



Check [www.vectorfabrics.com](http://www.vectorfabrics.com) for a free demo on concurrency analysis



# VectorFabrics

## Thank you!