

Application parallelization for multi-core Android devices

Klaas van Gend
klaas@vectorfabrics.com

LinuxCon / ELC Europe
Nov. 6, 2012, Barcelona, Spain



Let me introduce myself + Vector Fabrics



Name: Klaas van Gend

Founded: 1973

History / Claim to fame:

- Started programming C at 14
- First experience with Linux in 1993
- Co-founder of ELC-Europe

Name: Vector Fabrics

Founded: 2007

Claim to fame:

2012: Pareon tool •
analyse & optimize for multicore •



Multicore 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??

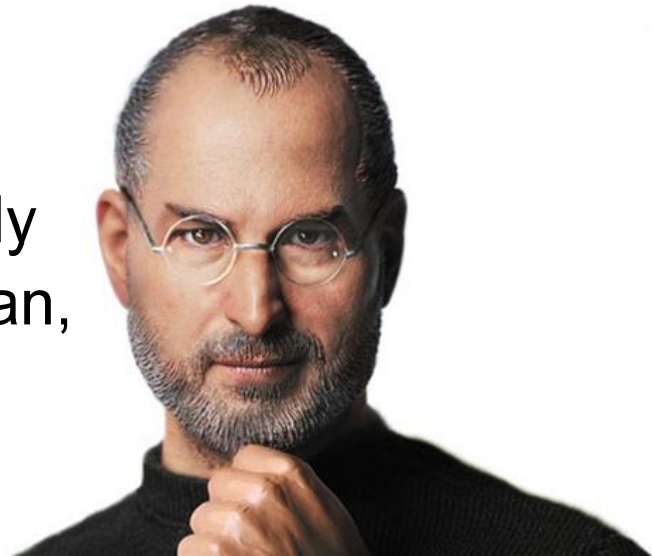
Creating parallel programs is hard...

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.”



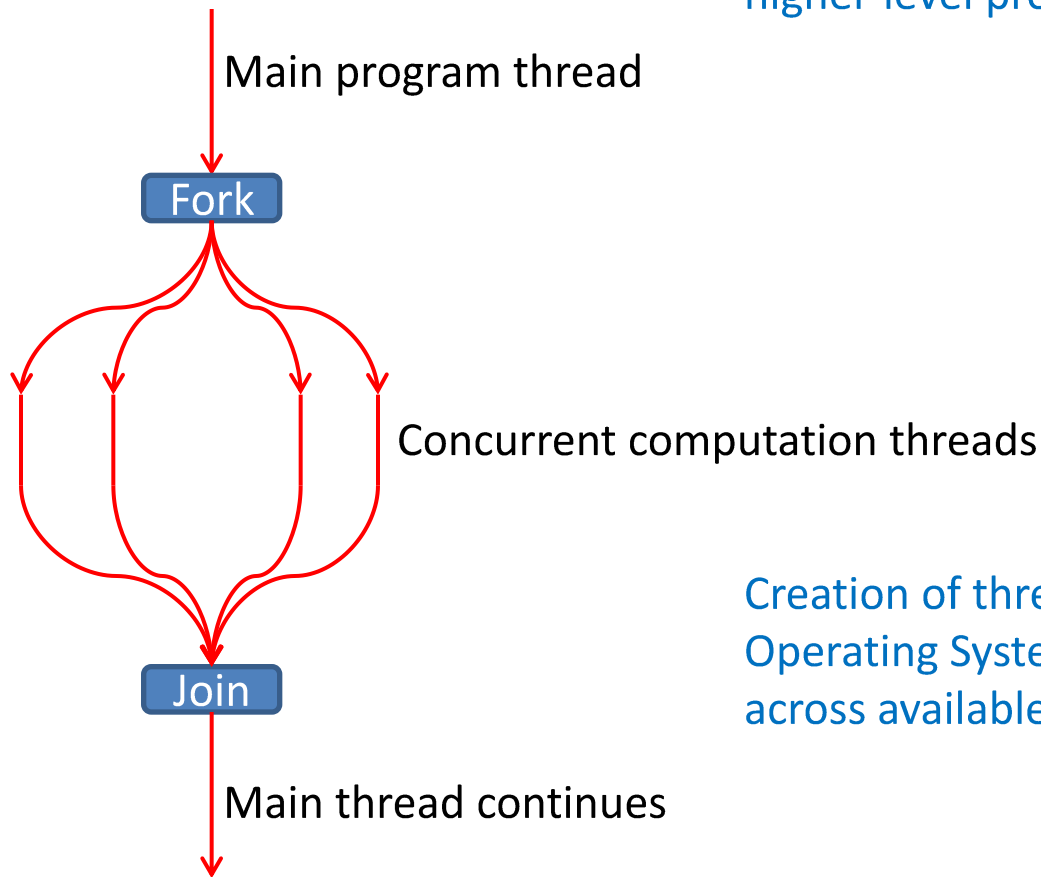
Presentation index

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

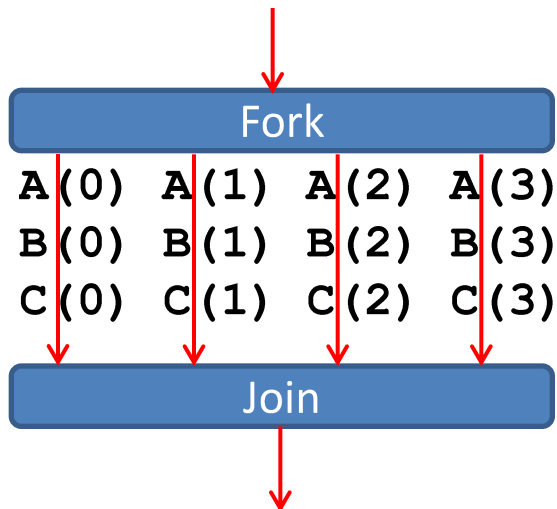
Source code:

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

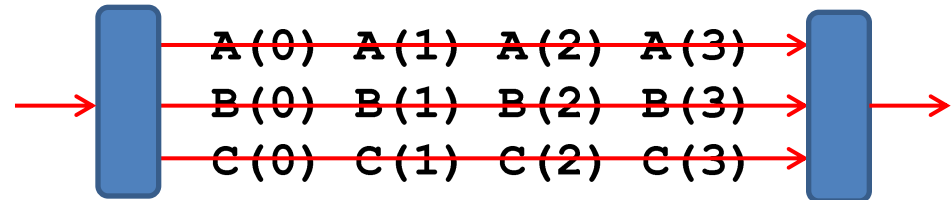
Sequential execution order:

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

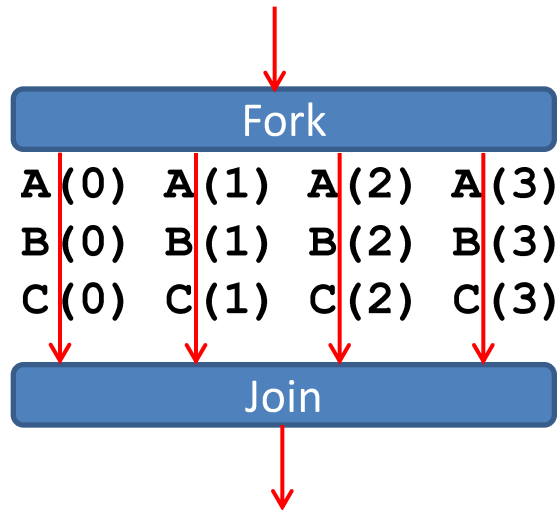
Data partitioning:



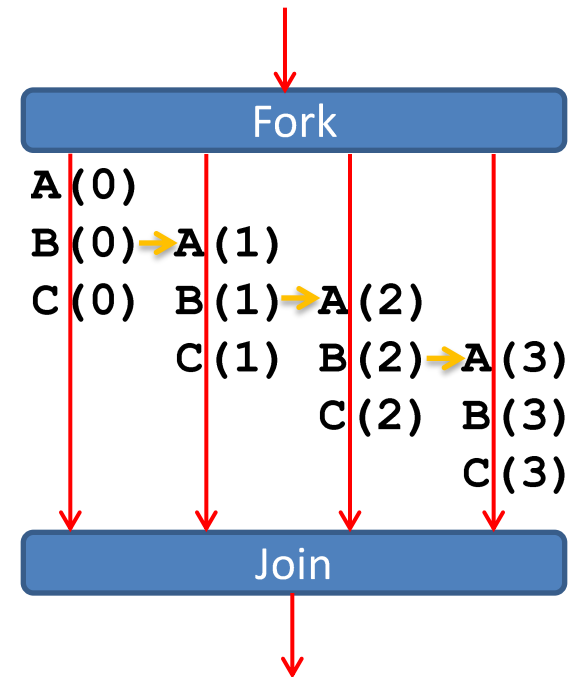
Task partitioning:



Issue: Data dependencies



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



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 Pareon 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: Parallelization Analysis – 1

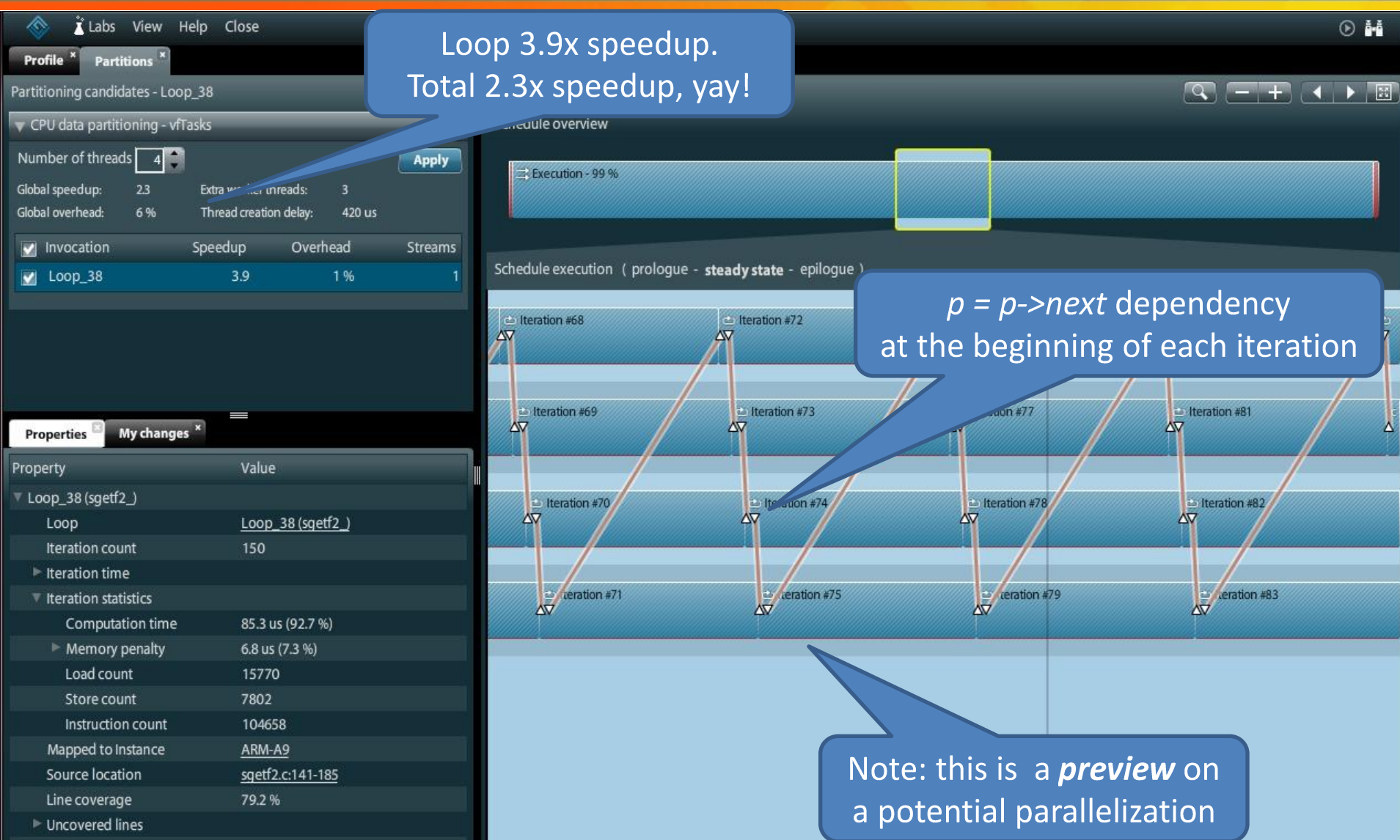
The screenshot displays the PAREON application interface with several panels and callouts:

- Properties Panel:** Shows memory dependency cluster 278 (carried by Loop_16429) with details like cluster context, dependencies, and memory dependency.
- Dynamic Help / Cheat sheets Panel:** Provides an introduction to the parallelization process, mentioning the help icon and check icon.
- Profile / Partitions Panel:** A table showing coverage and delay for various tasks and loops.
- 2D-Profile / Schedule / Architecture Panel:** A 3D visualization of the execution schedule showing tasks and loops as blocks.
- Callouts:**
 - Profiling info, like 'normal' profilers:** Points to the Properties panel.
 - Browsing through your application, 100% width means active 100% of run:** Points to the 2D-Profile panel.
 - Oops, there are dependencies between iterations of Loop_4179!** Points to the bottom section of the 2D-Profile panel.

Name	Coverage	Delay
Parallel_16428	97	94.06
Task_16429	97	94.06
Mat::ptr	100	0.00
Parallel_16430	100	27.44
Task_16431	100	27.43
Loop_4178	100	0.34
_vf_memset	0	0.00
Loop_4179	100	48.09
Loop_4215		
Loop_4216		
Parallel_16432		
Loop_4217		
Loop_4218		
Loop_4219		
Loop_4220		
Loop_4221		
Loop_4222		
Loop_4223		
Loop_4224		
Loop_4225		
Loop_4226		
Loop_4227		
Loop_4228		
Loop_4229		
Loop_4230		
Loop_4231		
Loop_4232		
Loop_4233		
Loop_4234		

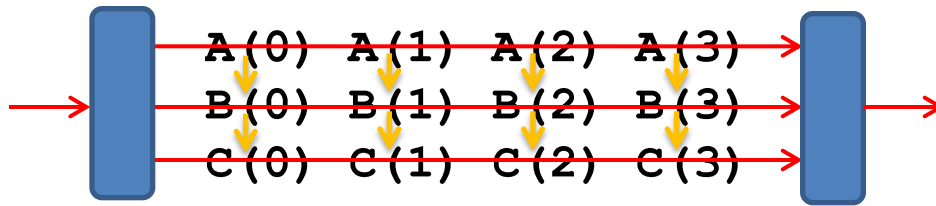
16429 total loop carried transfer rate: 8.6 Mi transfers/s
(0.0 transfers/s); 6 data dependency clusters (8.6 Mi transfers/s);
dependencies (18.0 transfers/s); 92 anti- and output dependency clusters

PAREON: Parallelization Analysis - 2

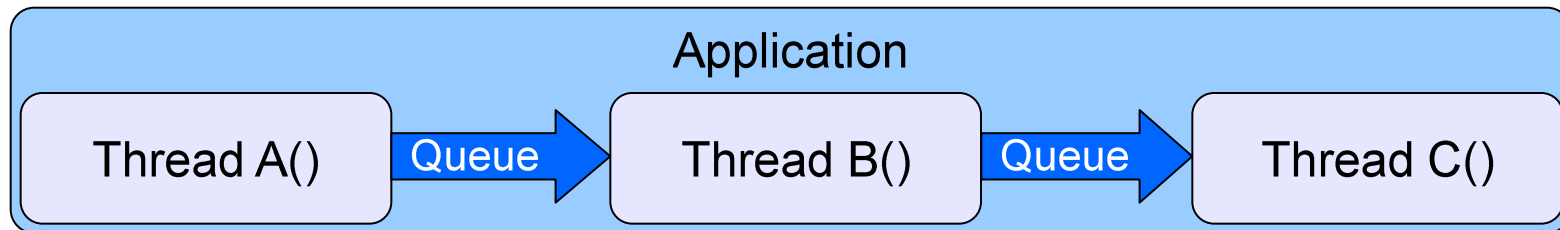


Pipelining: Data deps & functional partitioning

Functional 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!

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

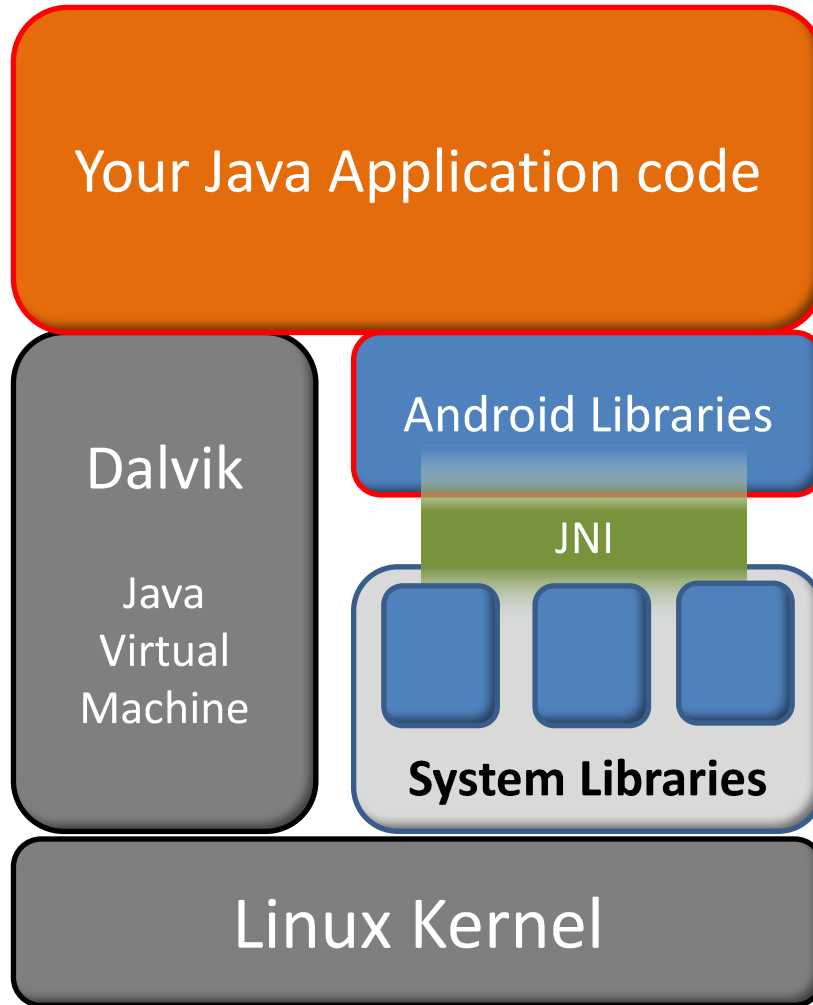
Managable development process!

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**



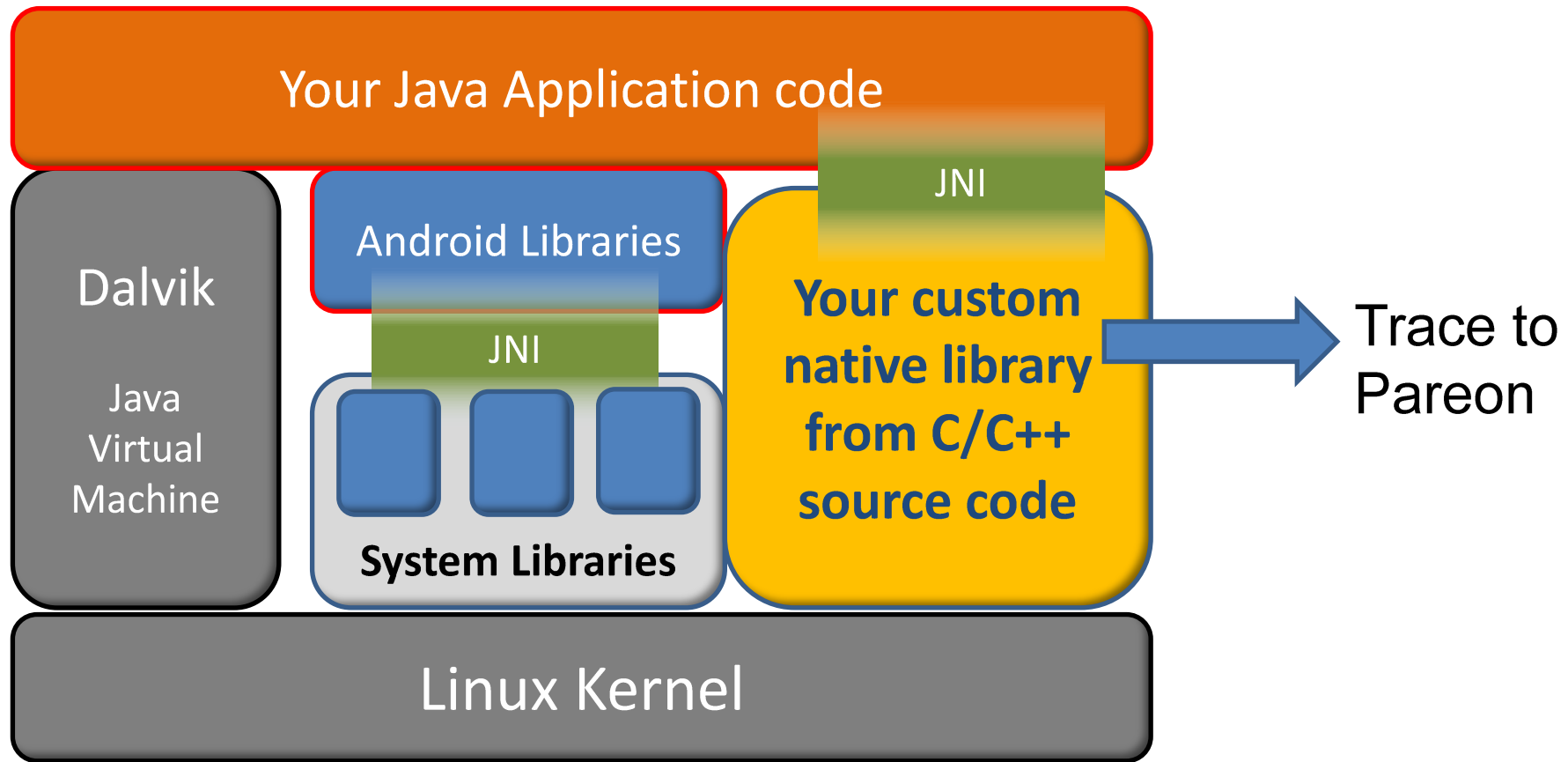
Android Application 1: Plain, just Java



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

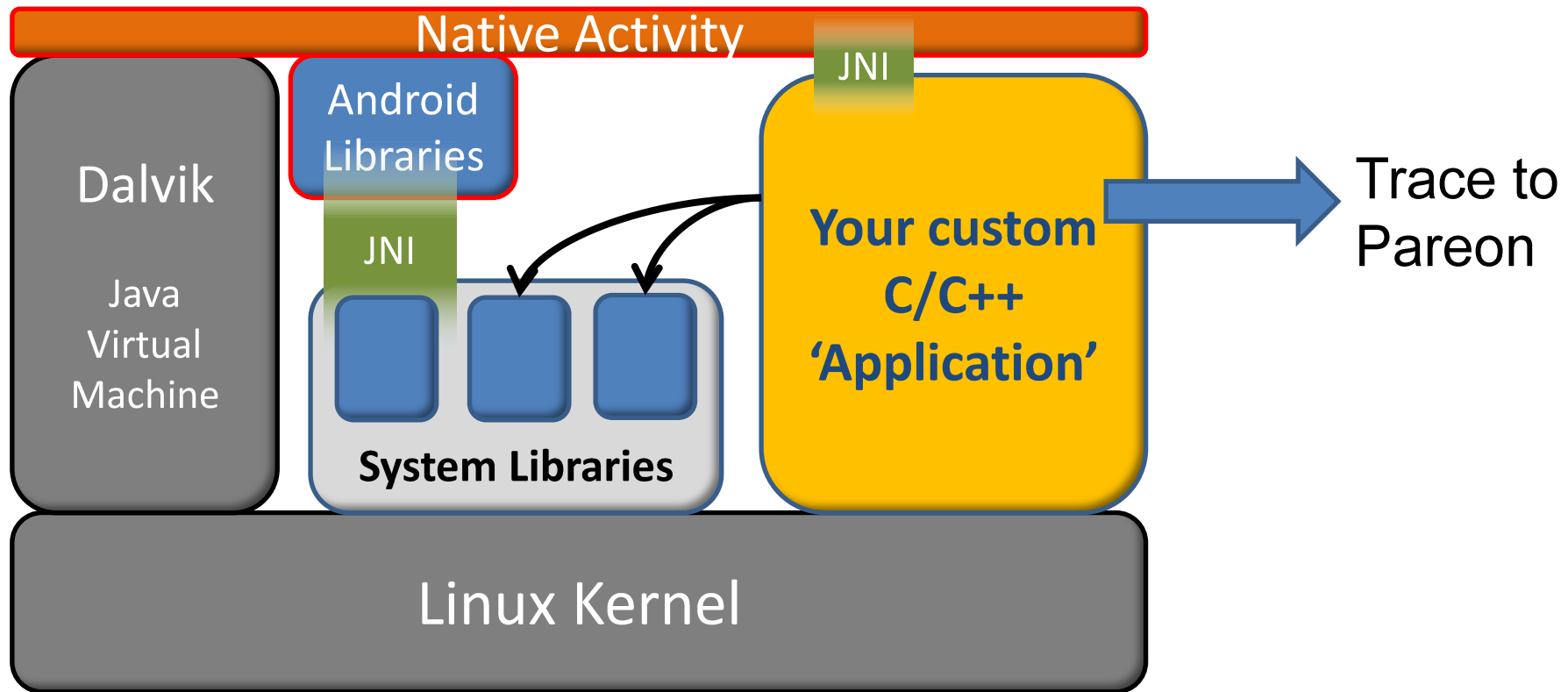
Android Application 2: with native libraries

Apps can include “native” binary code for best performance

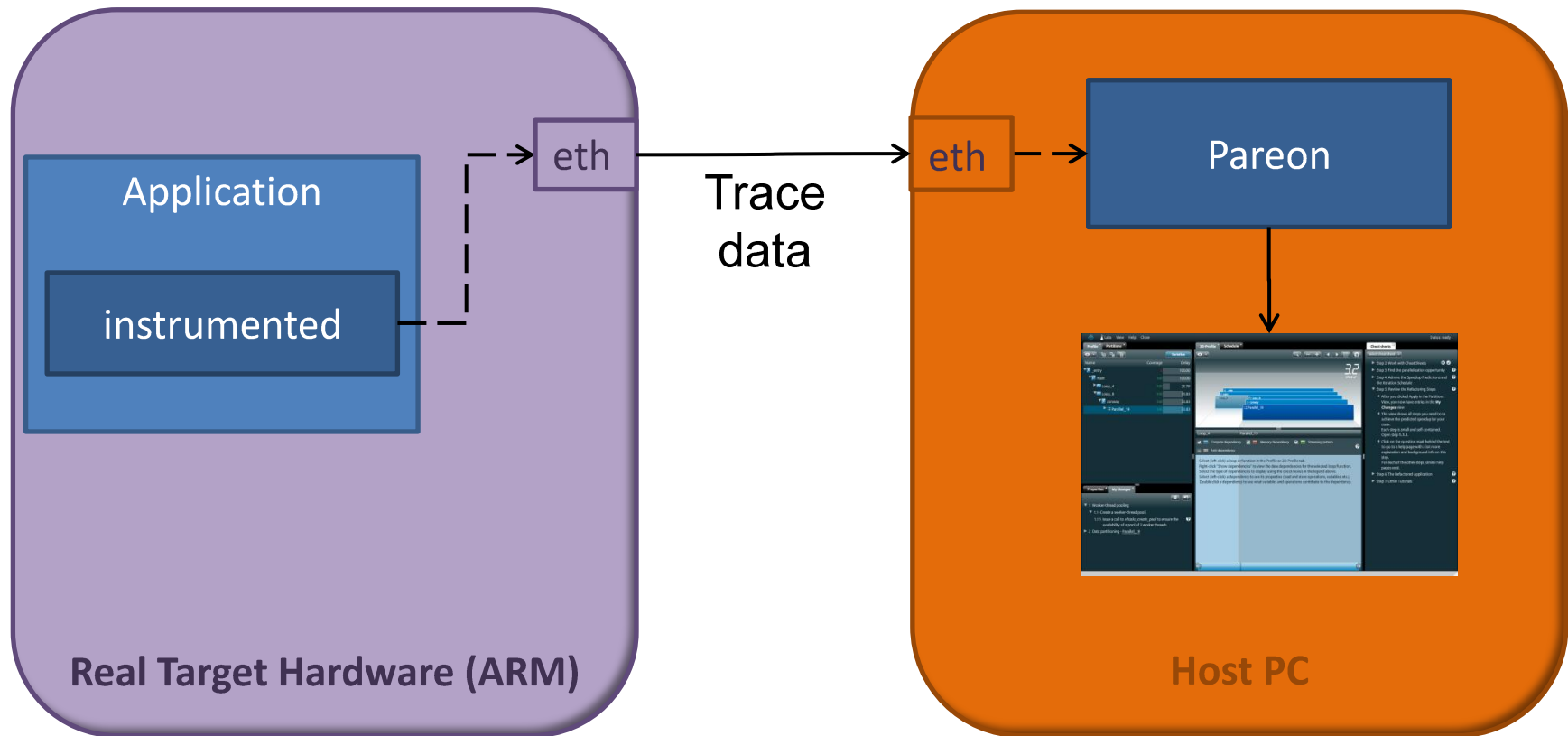


Android Application 3: NativeActivity

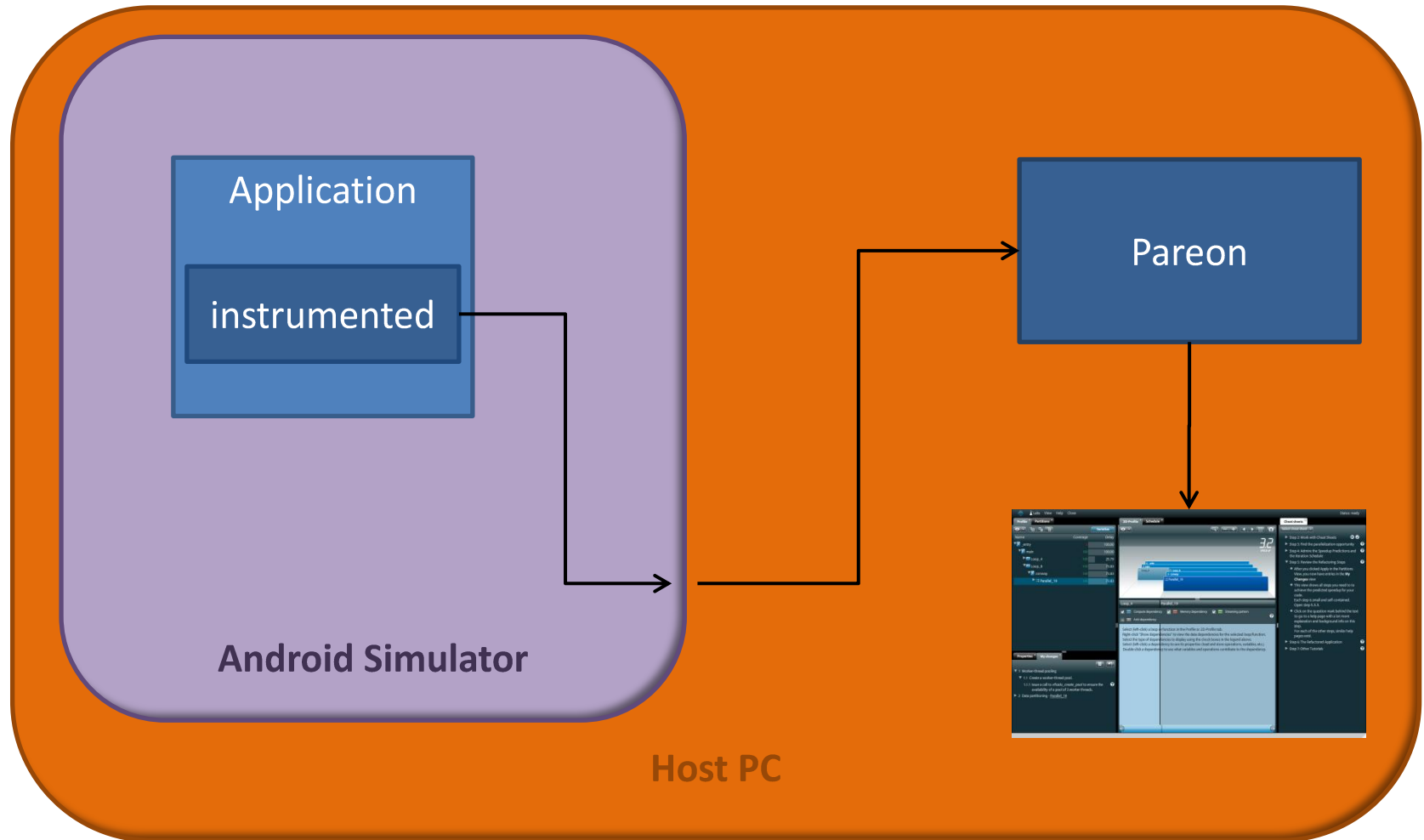
“Native activities” are created without Java source code



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 interface, which is used for analyzing and optimizing code for data parallelism. The interface is divided into several panels:

- Profile / Partitions:** This panel shows partitioning candidates for Loop_313. It includes a table of CPU data partitioning results.
- Properties / My changes:** This panel provides detailed information about the selected partitioning candidate, Loop_313.
- 2D-Profile / Schedule / plasma.c:** This panel shows a 2D profile of the code, highlighting the execution of Loop_313. It includes a speedup indicator and a visual representation of the code's execution flow.

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 / 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

Speedup: 1.0

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

- Profile / Partitions:** This panel shows partitioning candidates for 'Loop_313'. It includes a table of candidates and a table of properties.
- 2D-Profile / Schedule:** This panel shows a 2D profile of the code, with a red circle highlighting the 'Loop_313' region.
- Properties / My changes:** This panel shows the properties of the selected partition, including its compute dependency, source, and destinations.

Partitioning candidates - Loop_313

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

The 2D profile shows a series of blue bars representing different code regions. A red circle highlights the 'Loop_313' region. The 'Schedule' panel shows the 'Loop_313' region with a red circle around it. The 'Schedule' panel also shows the 'Loop_313' region with a red circle around it.

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

- Profile / Partitions:** This panel shows partitioning candidates for `Loop_313`. It includes a section for CPU data partitioning with a "Number of threads" set to 4 (highlighted with a red circle). Below this, a table lists partitioning candidates with their respective speedup and overhead.
- Properties / My changes:** This panel shows the properties of the selected partitioning candidate, `Loop_313`. It includes a table with properties such as "Compute dependency", "Ignore with data partitioning", "Source", "Operation (+)", "Location", "Destinations", "#loop-carried transfers", and "Loop carried".
- 2D-Profile / Schedule:** This panel shows a 2D profile of the code, with a focus on `Loop_313`. It includes a "Schedule" tab and a "plasma.c" file. The profile shows the execution of `Loop_313` and its dependencies, with a "SPEED-UP" indicator showing a value of 1.0.

Table 1: Partitioning candidates - Loop_313

Partitioning candidates - Loop_313	Speedup	Overhead	Streams
Loop_120	4.0	1 %	0
Loop_189	4.0	1 %	0
Loop_313	3.9	1 %	0

Table 2: Properties of Loop_313

Property	Value
Compute dependency	136.46
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

Table 3: Loop_313 total loop carried transfer rate

Transfer Rate
136 Ki transfers/s

Table 4: Loop_313 total loop carried transfer rate breakdown

Transfer Rate
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

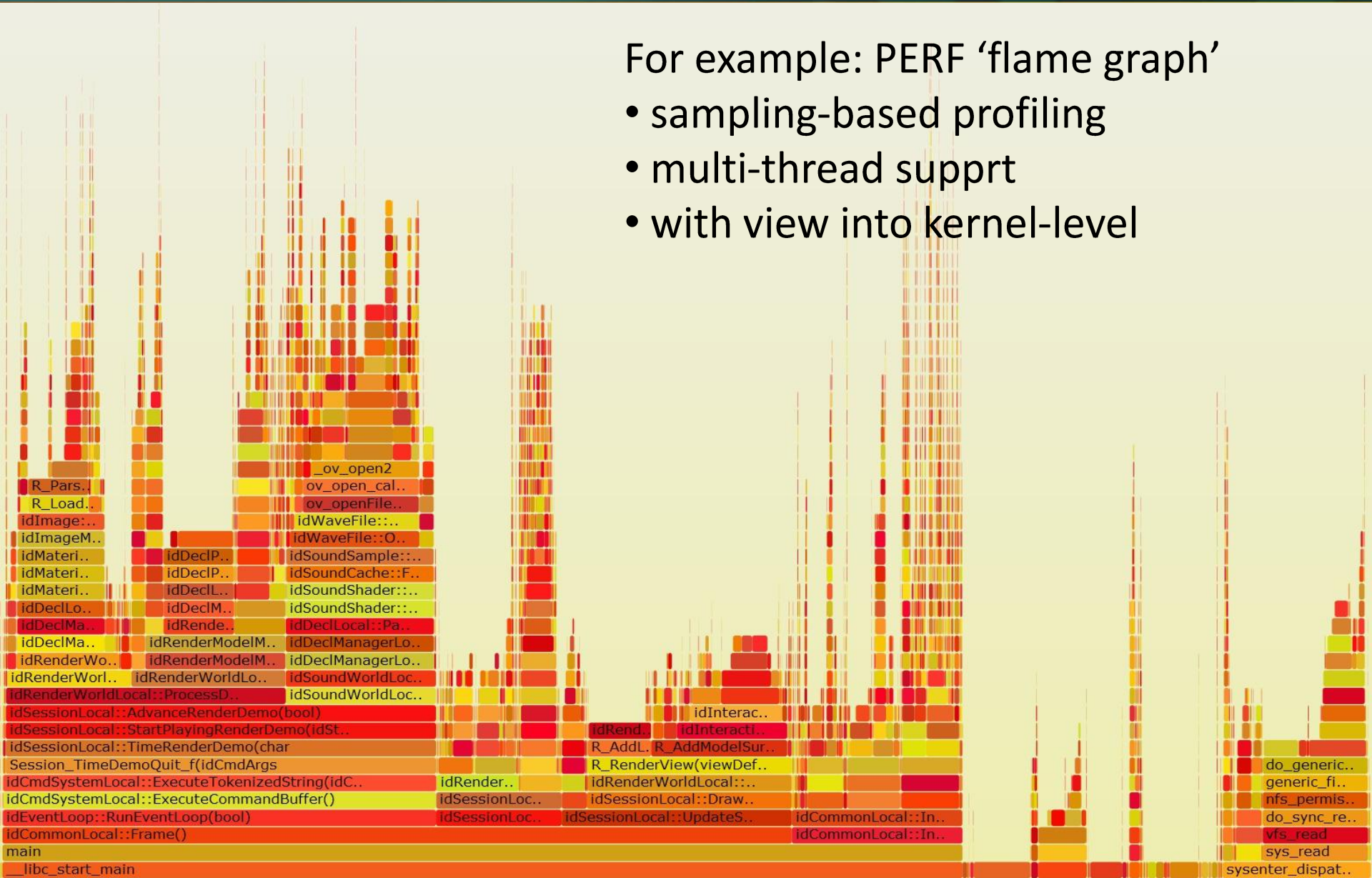
Not parallelized: JNI call to render frame



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

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



Check www.vectorfabrics.com for a free demo on concurrency analysis



VectorFabrics

Thank you!