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 Experia 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??

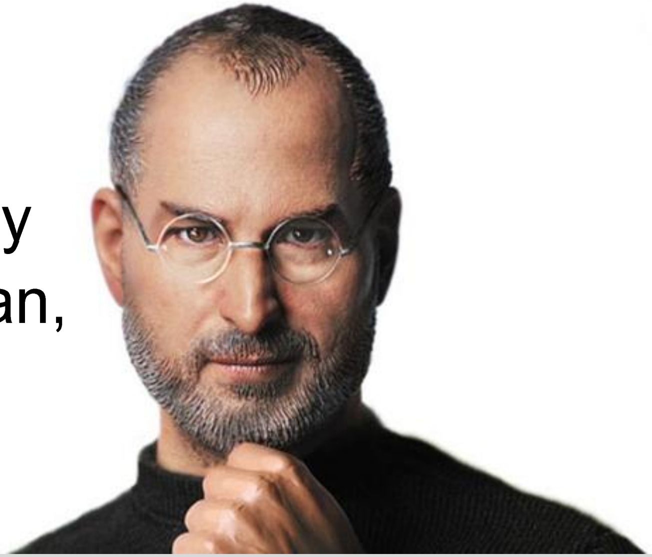
Creating parallel programs is hard…

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.”

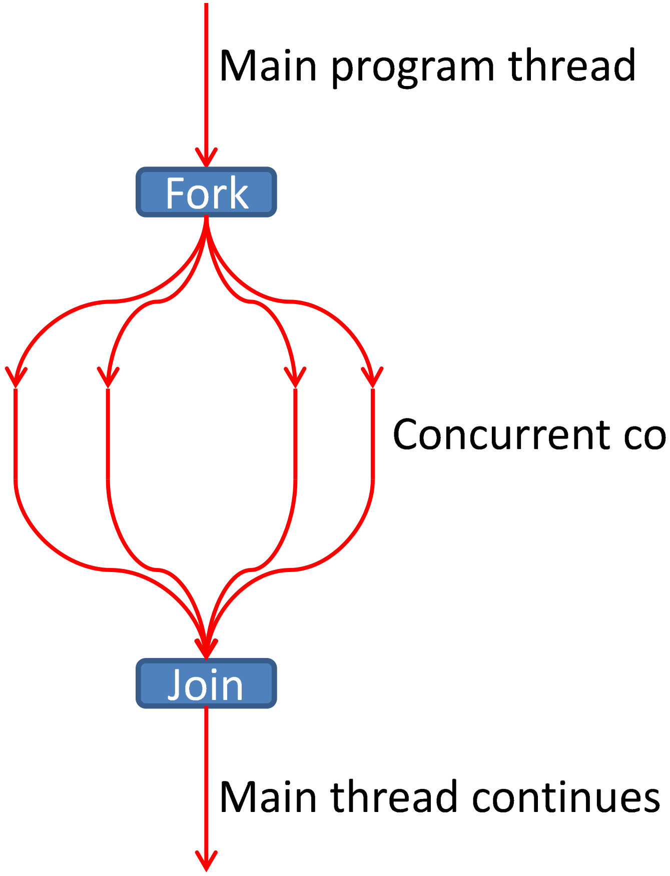


Presentation index

- 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



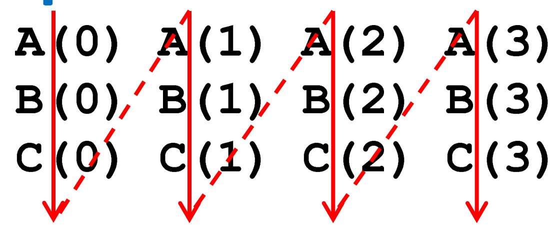
Creation of threads is application responsibility. Operating System handles run-time scheduling across available processors!

Parallelization – two partitioning options

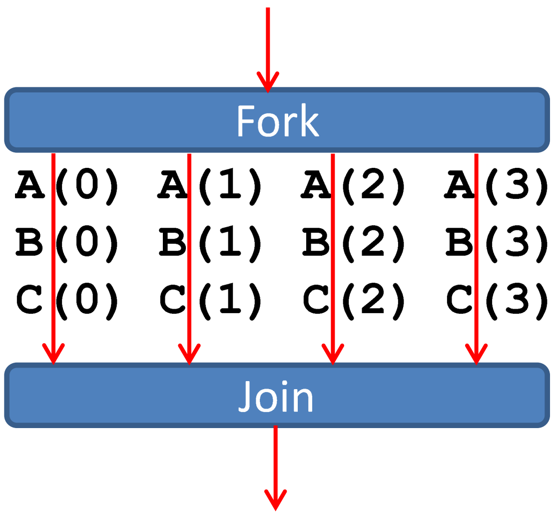
Source code:

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

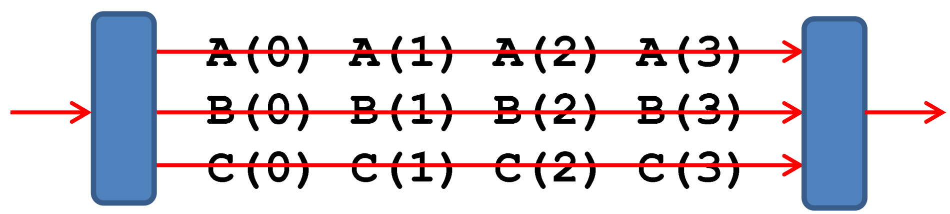
Sequential execution order:



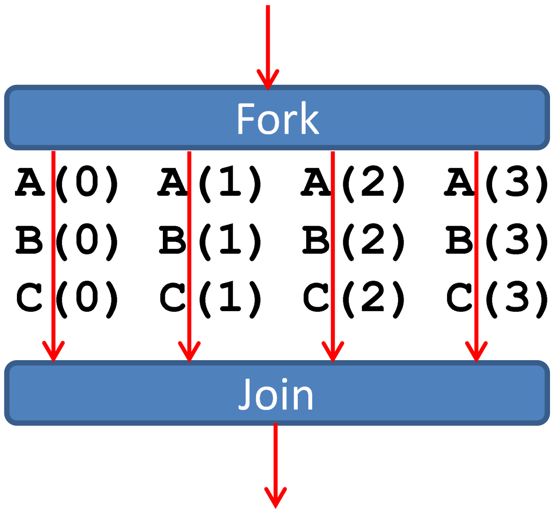
Data partitioning:



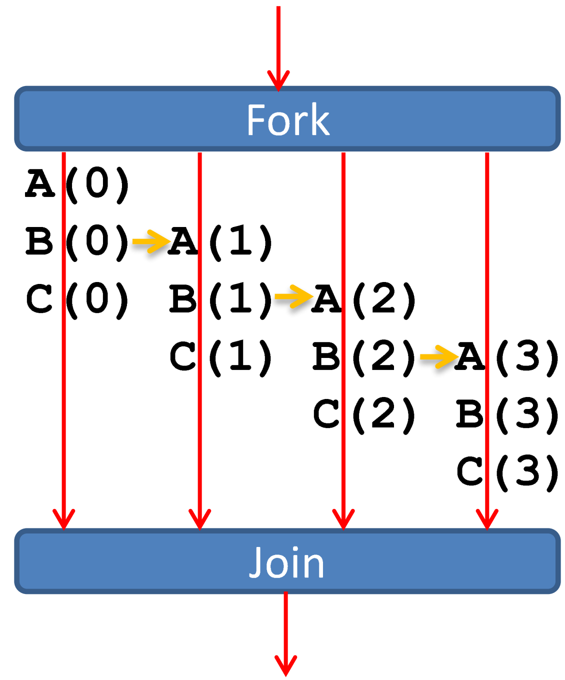
Task partitioning:



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-parrallel 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

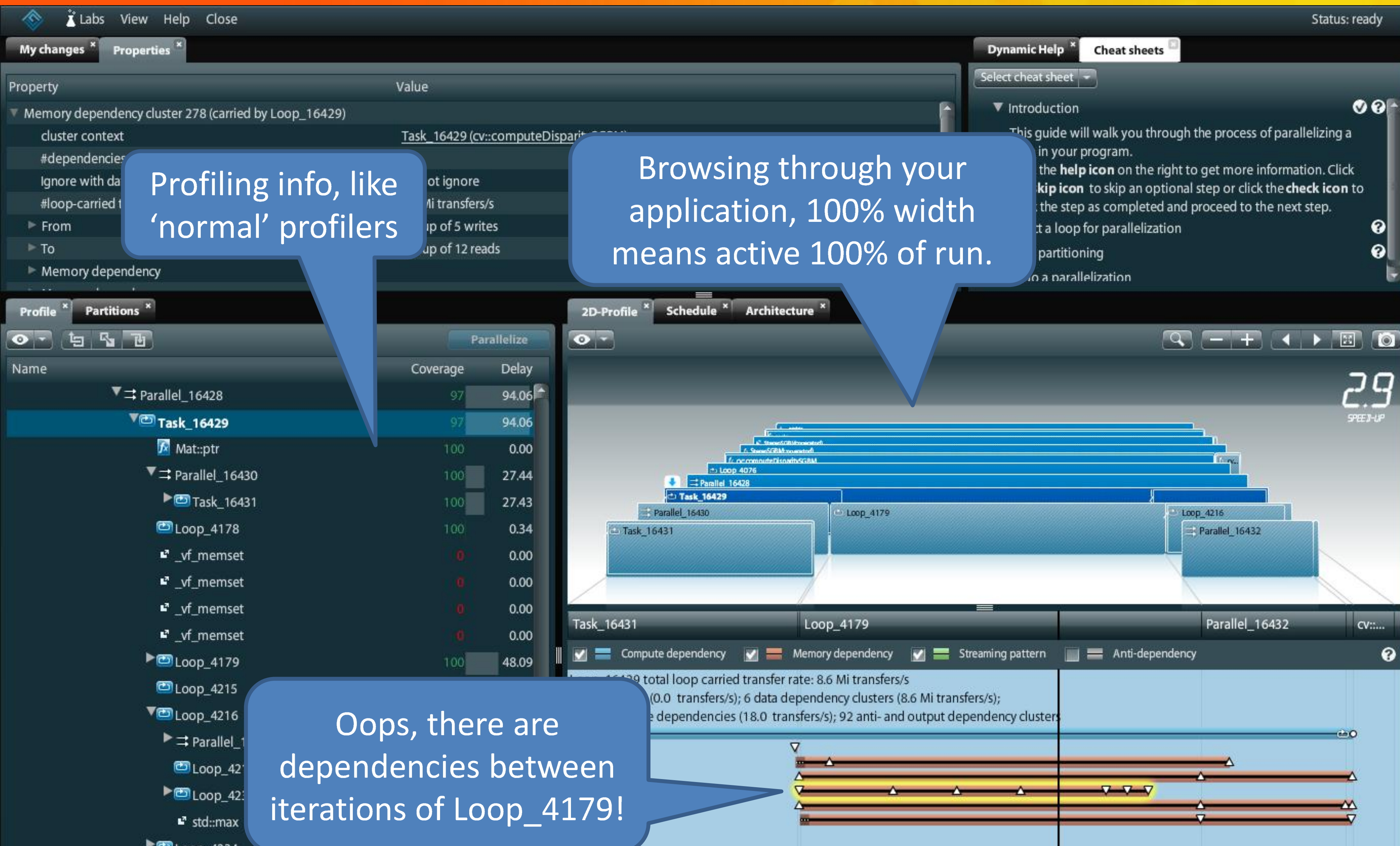
```
// conditionally accumulate results
int acc = 0;
for (i=0; i<N; i++)
{
  int result = some_work(i);
  if (some condition(i))
    acc += result;
}
...use of acc ...
```

- 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

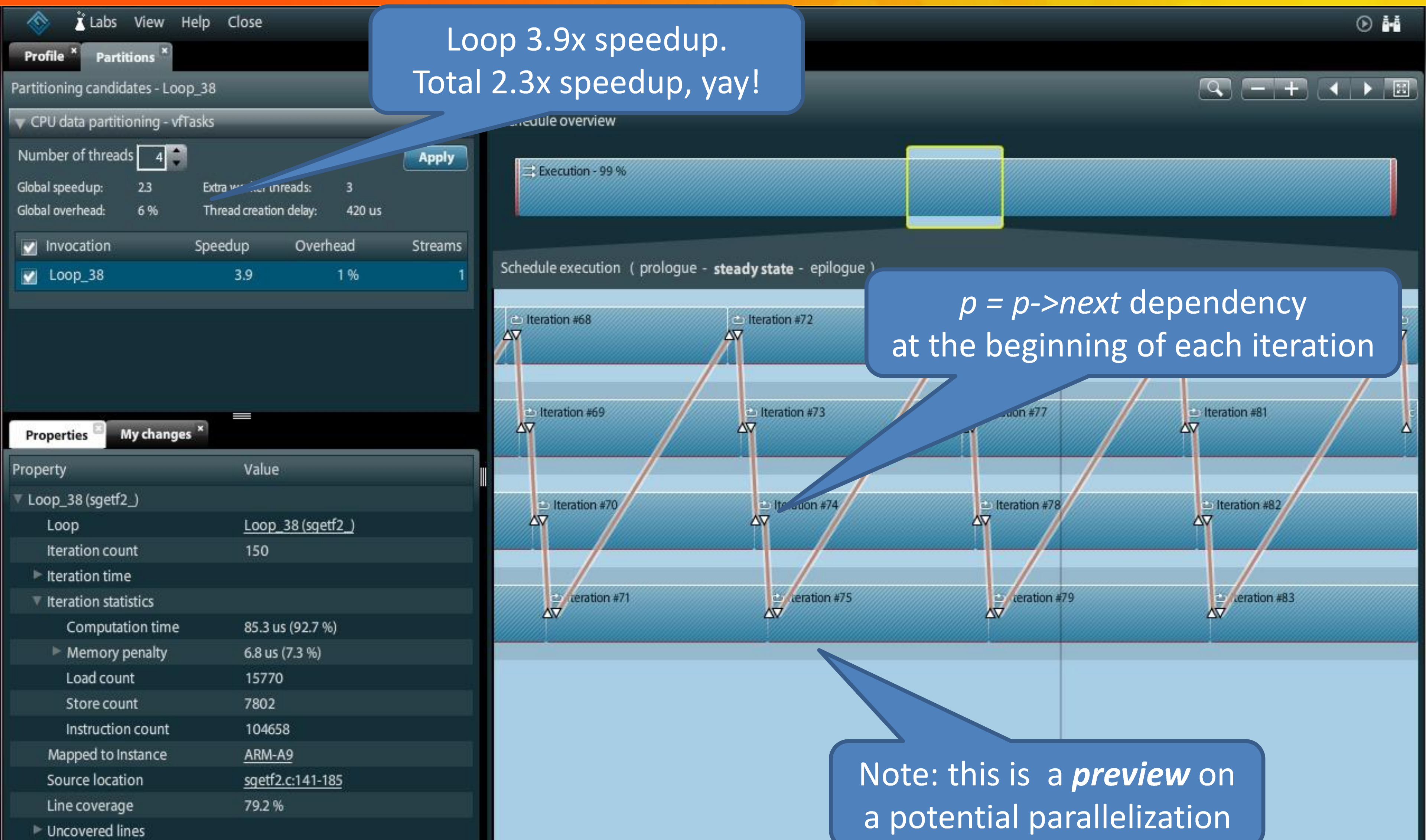


Profiling info, like 'normal' profilers

Browsing through your application, 100% width means active 100% of run.

Oops, there are dependencies between iterations of Loop_4179!

PAREON: Parallelization Analysis - 2



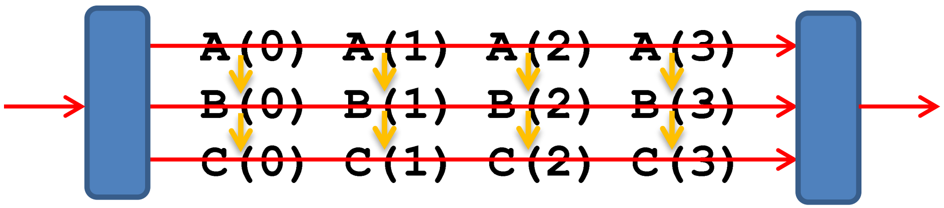
Loop 3.9x speedup.
Total 2.3x speedup, yay!

p = p->next dependency
at the beginning of each iteration

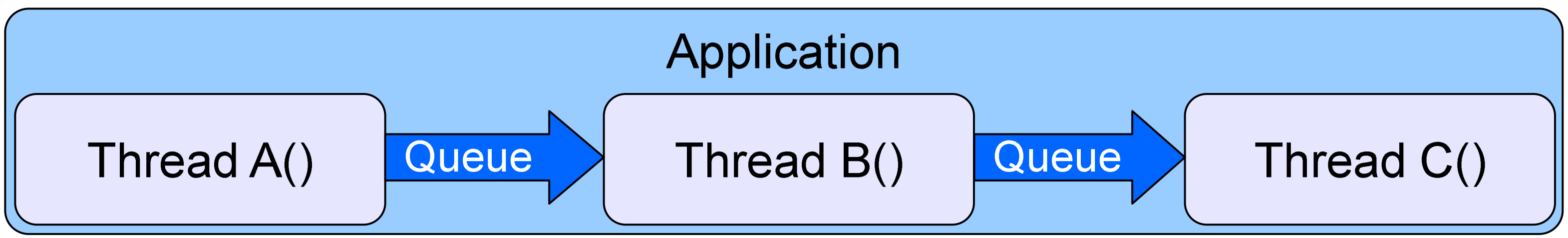
Note: this is a ***preview*** on
a potential parallelization

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 catched 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:

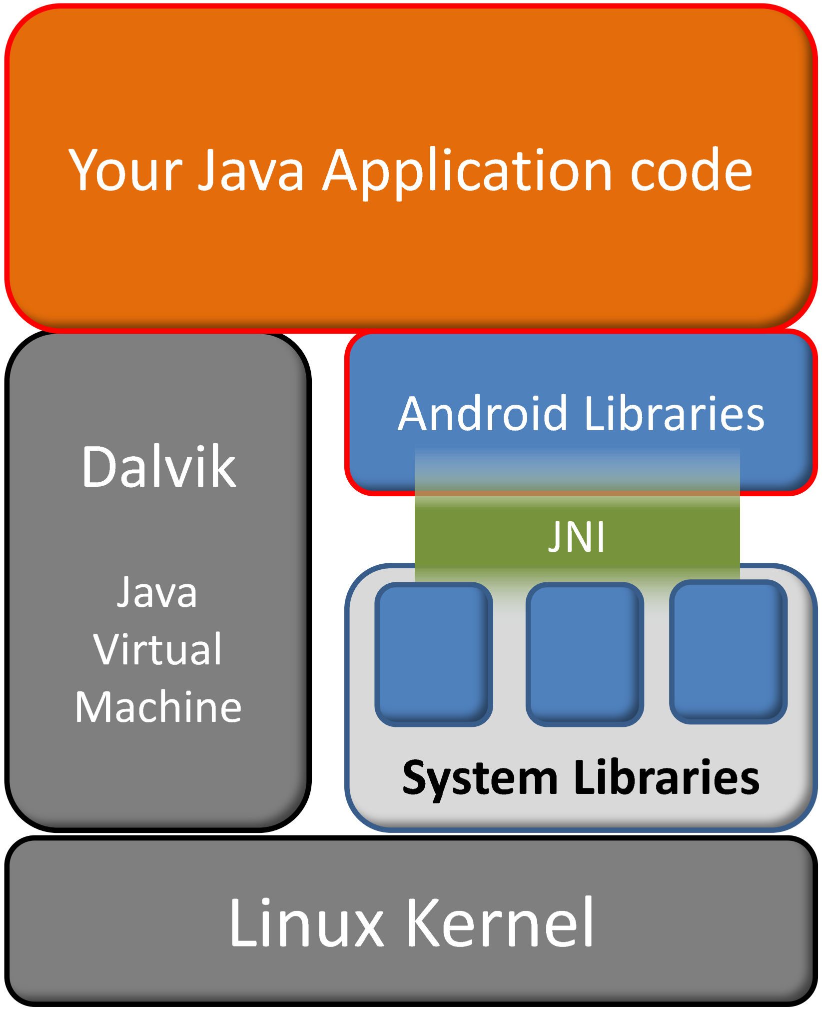
- Base upon a sequential program:
  functional and performance reference
- Apply higher-level parallelization patterns and primitives:
  clear semantics, re-use code, reduce risk
- Use tooling for analysis and verification
  - Prevent introduction of hard-to-find bugs
  - Prevent recoding effort that 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

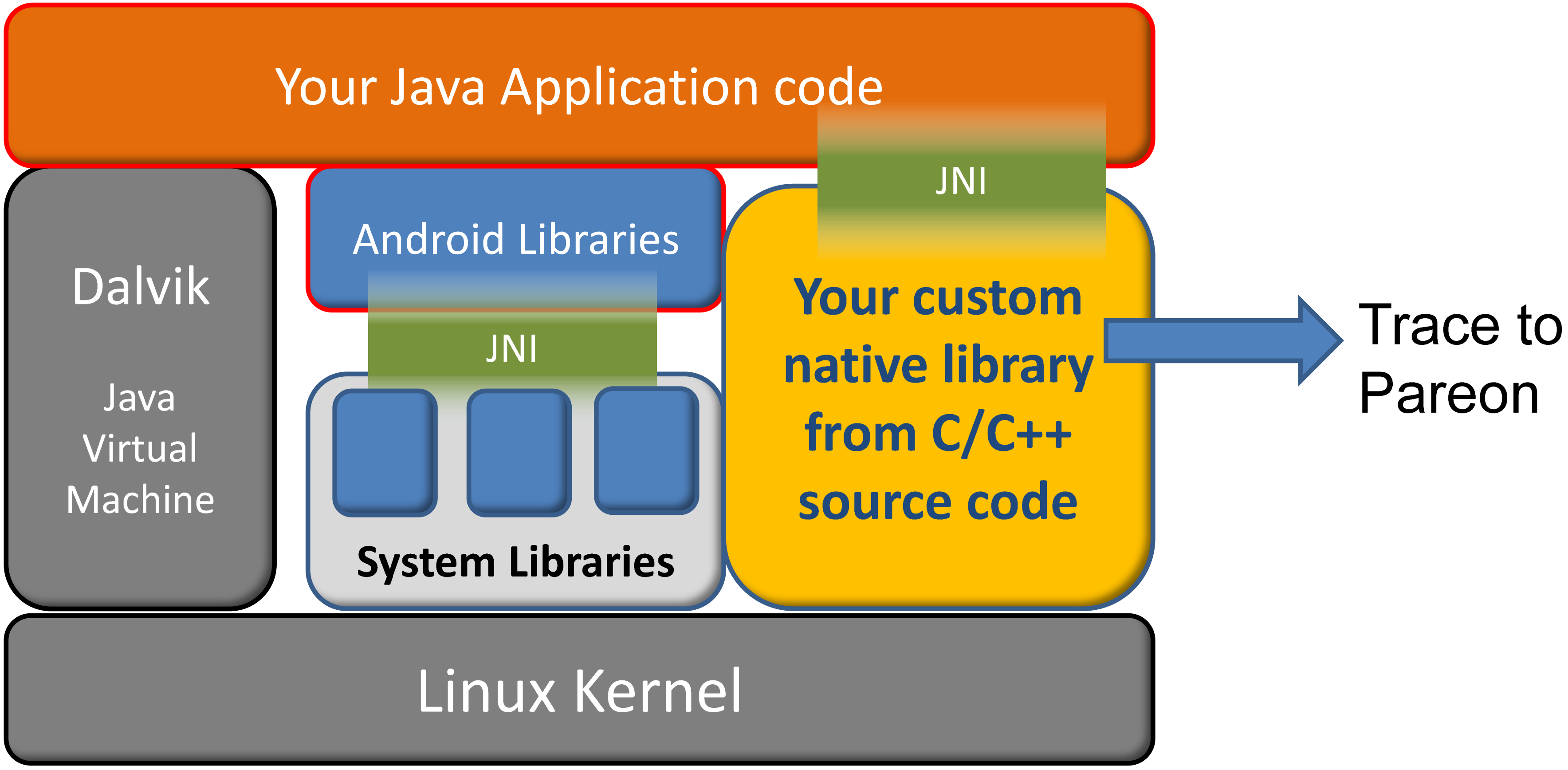
Android Application 1: Plain, just Java



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

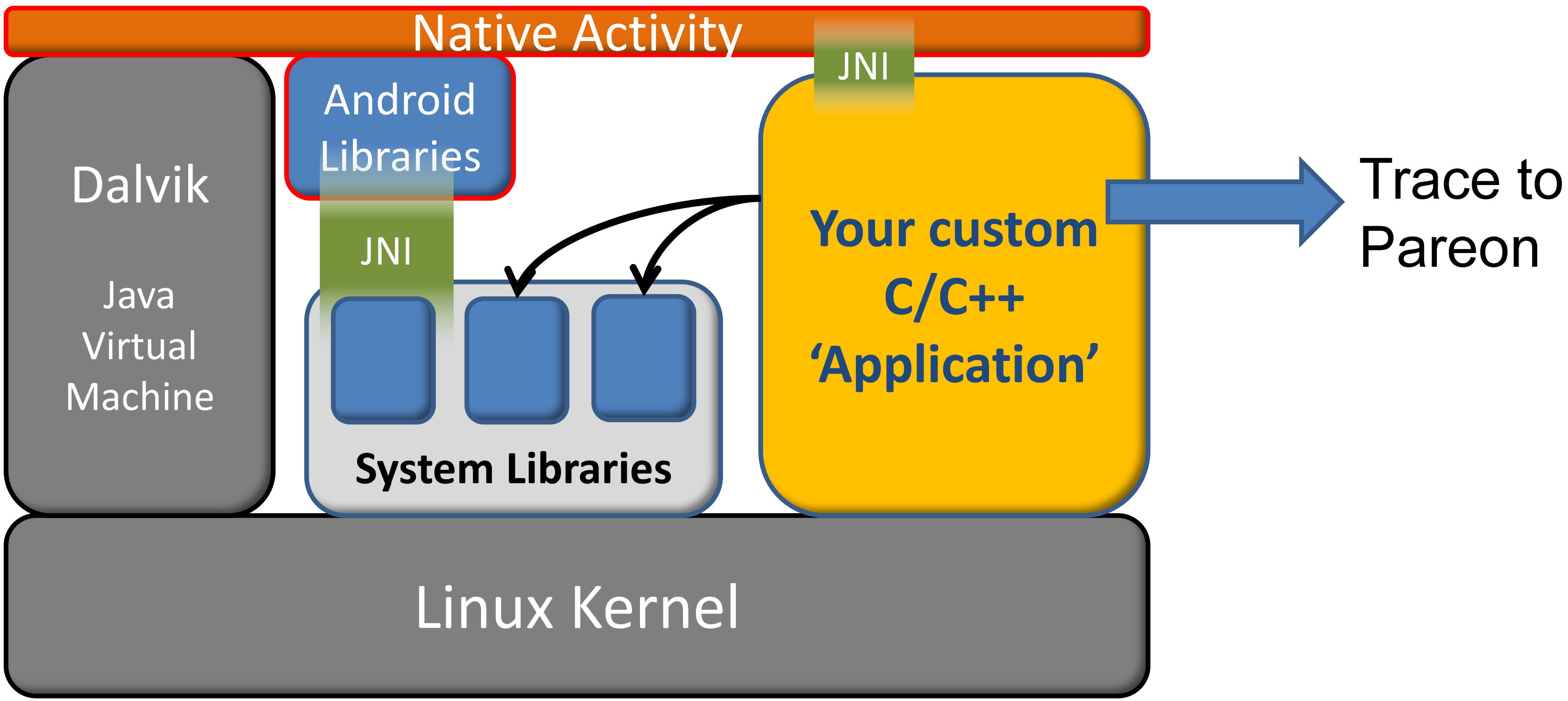
Android Application 2: with native libraries

Apps can include “native” binary code for best performance

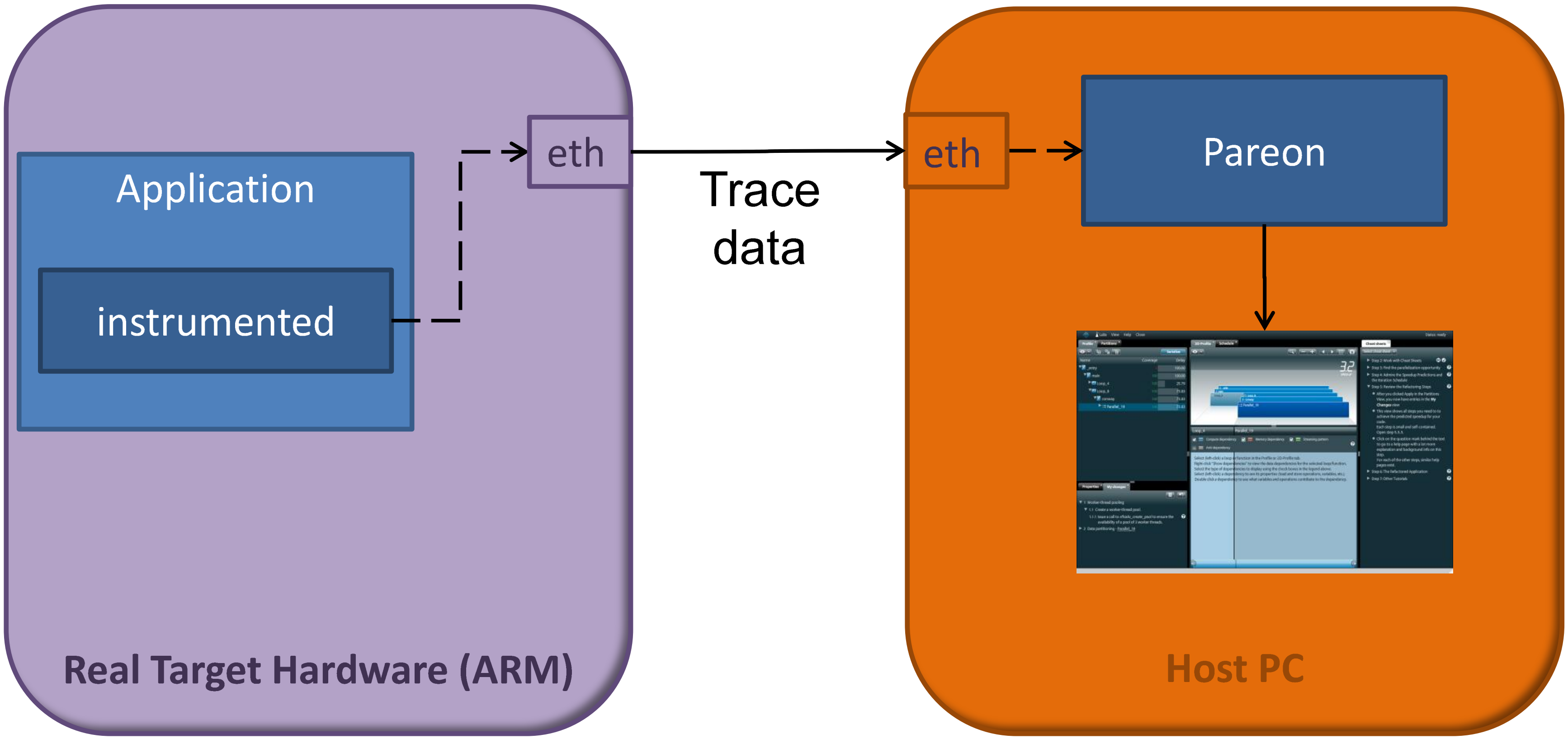


Android Application 3: NativeActivity

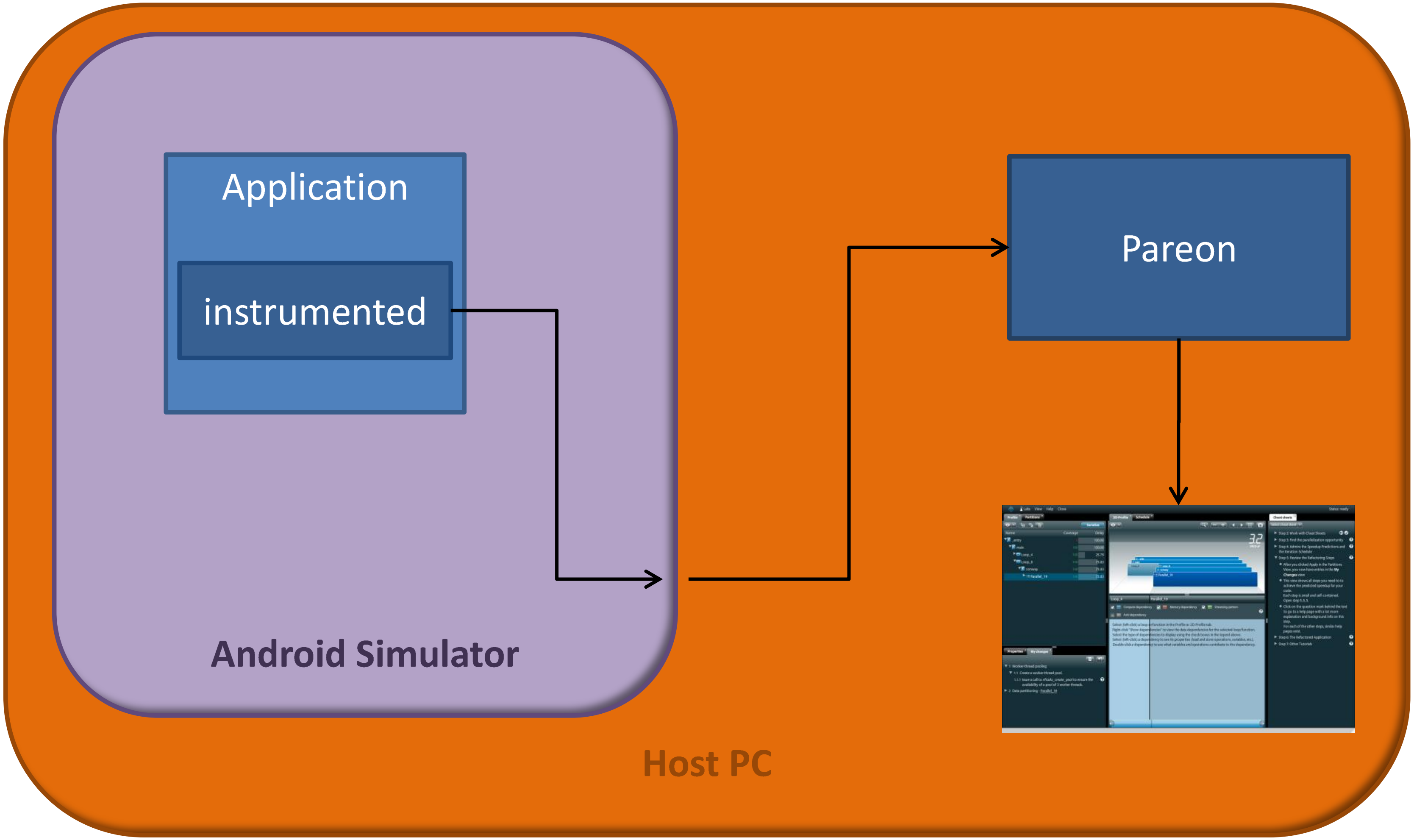
"Native activities" are created without Java source code



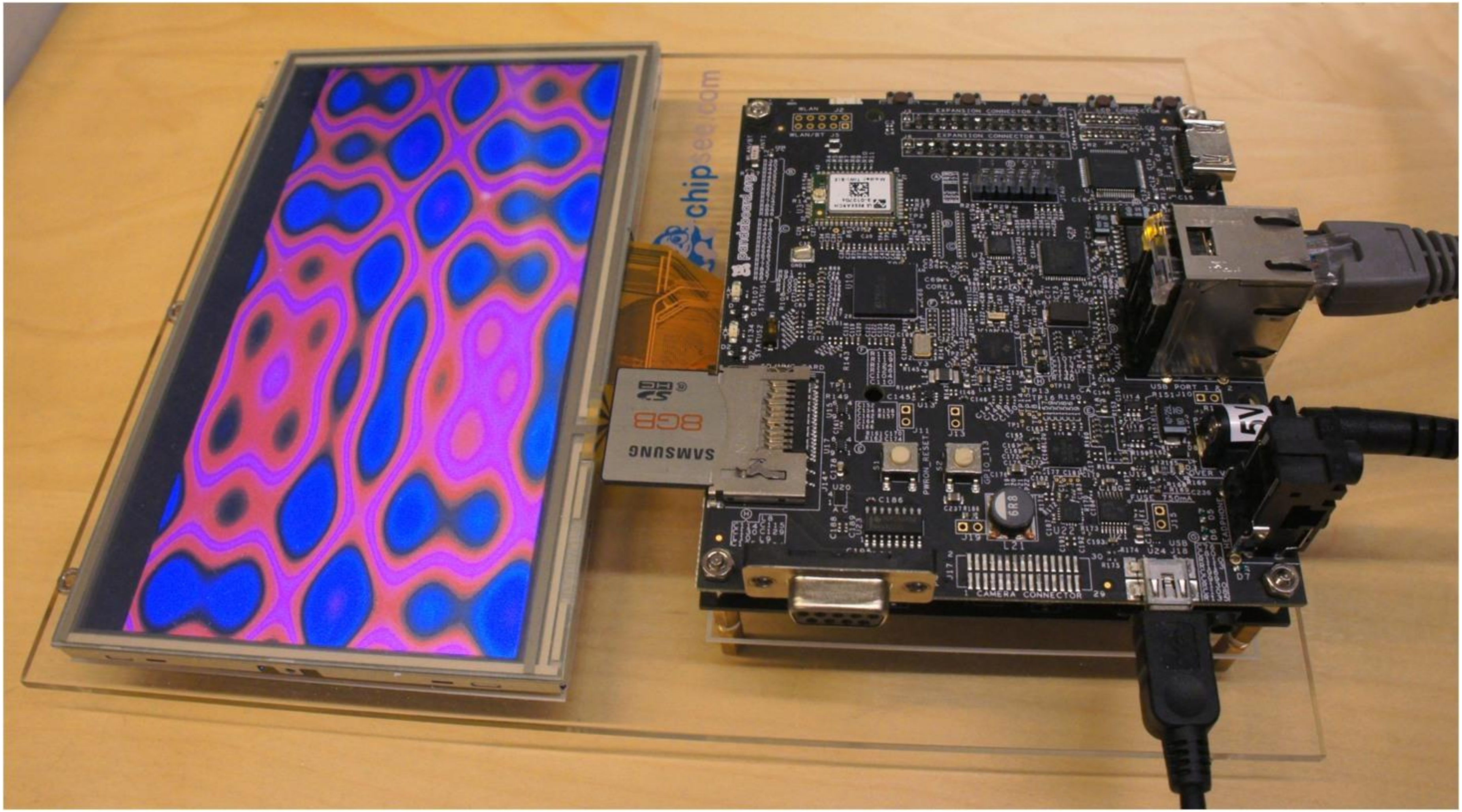
Application Analysis on Android target



System Setup using Android Simulator

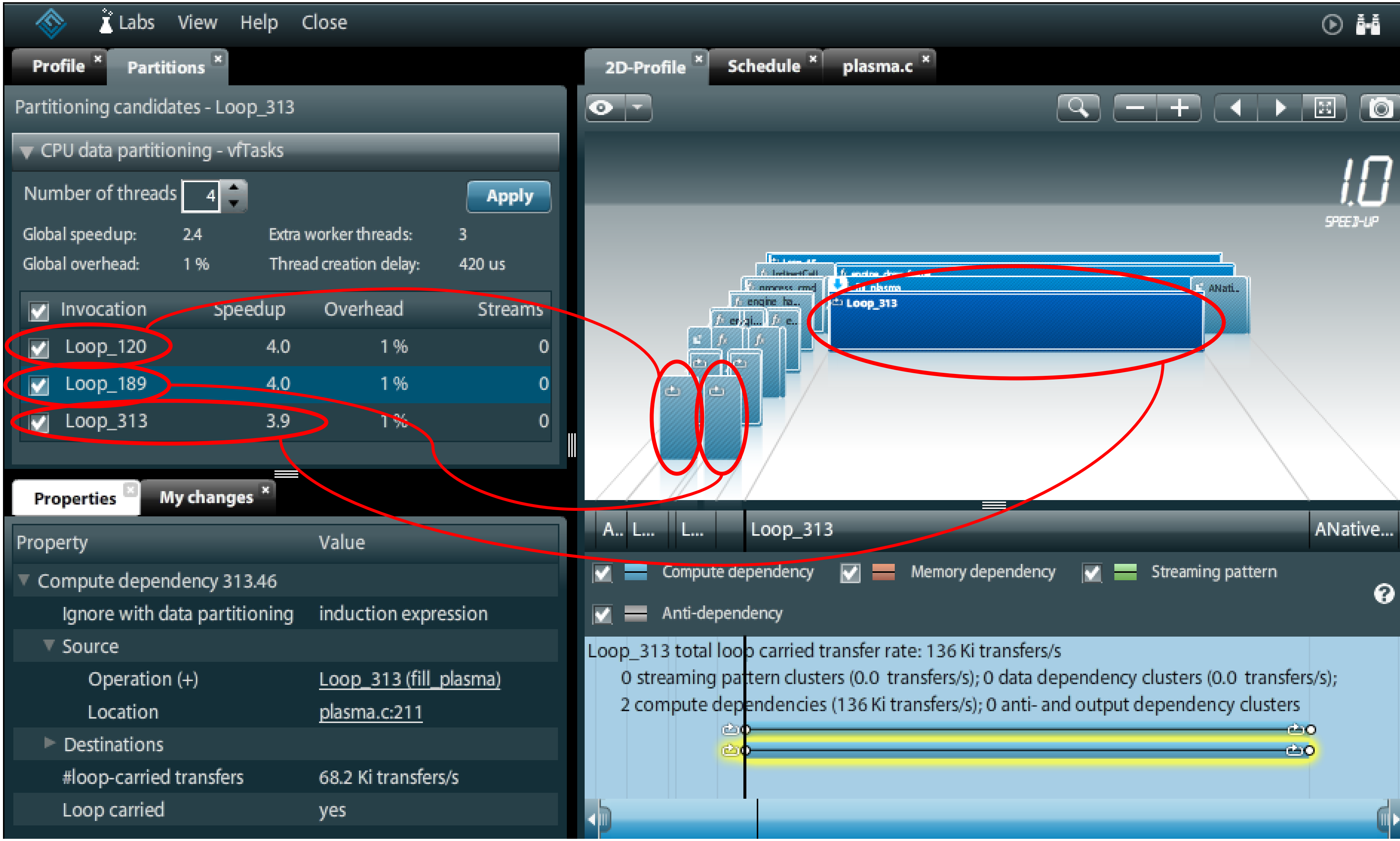


NDK plasma demo app analyzed on Android



Finding data parallelism on Android

Finding data parallelism on Android

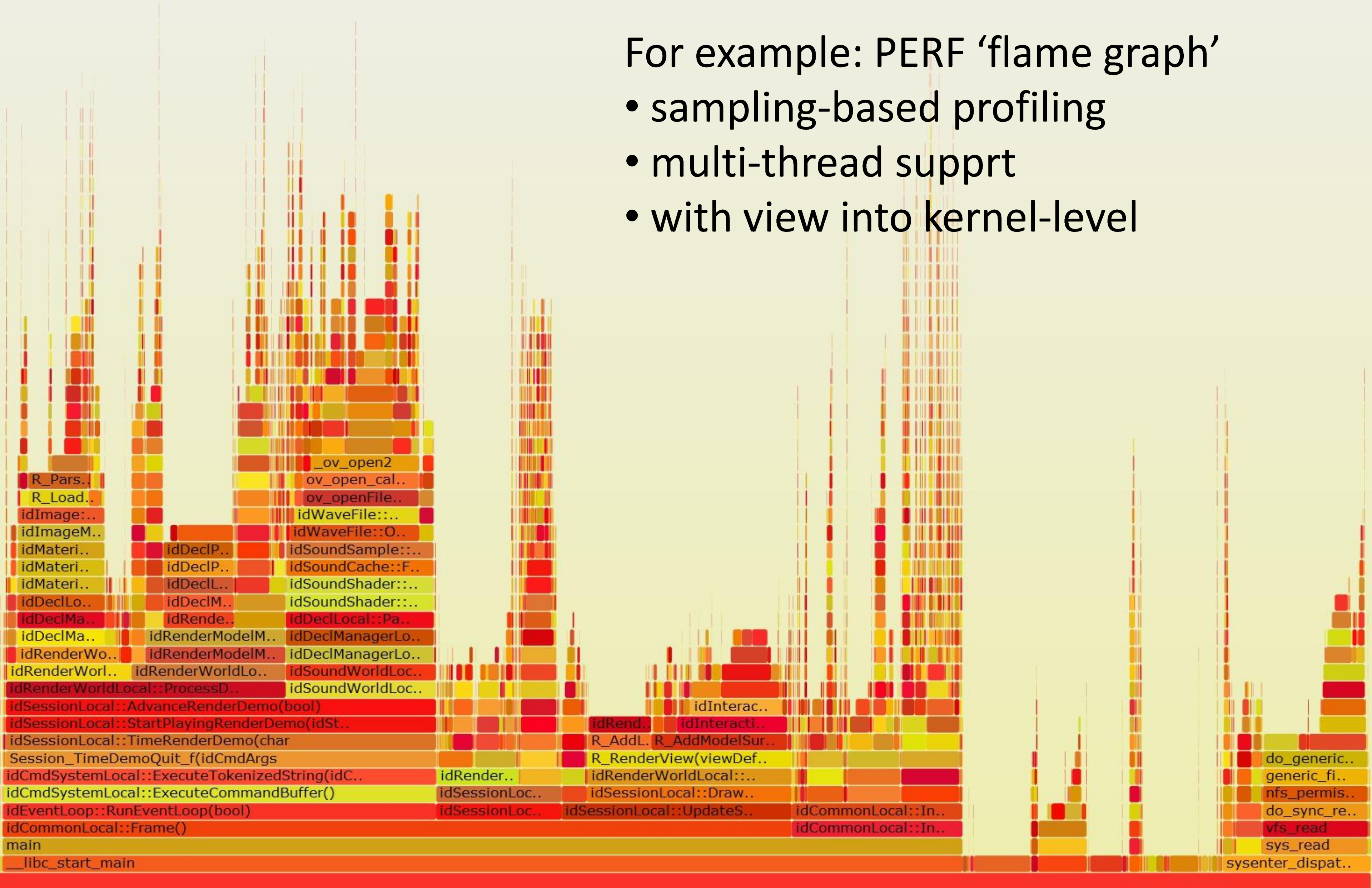


Finding data parallelism on Android

Not parallelized: JNI call to render frame

Performance Verification

For example: PERF ‘flame graph’
- sampling-based profiling
- multi-thread supprt
- 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 then 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 lower then 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!