BENCHMARKING OF I/O ACTIVITIES IN HPC APPLICATIONS WITH STRACE INSPECTOR

34TH POP WEBINAR | JUNE 2, 2025 | ARAVIND SANKARAN

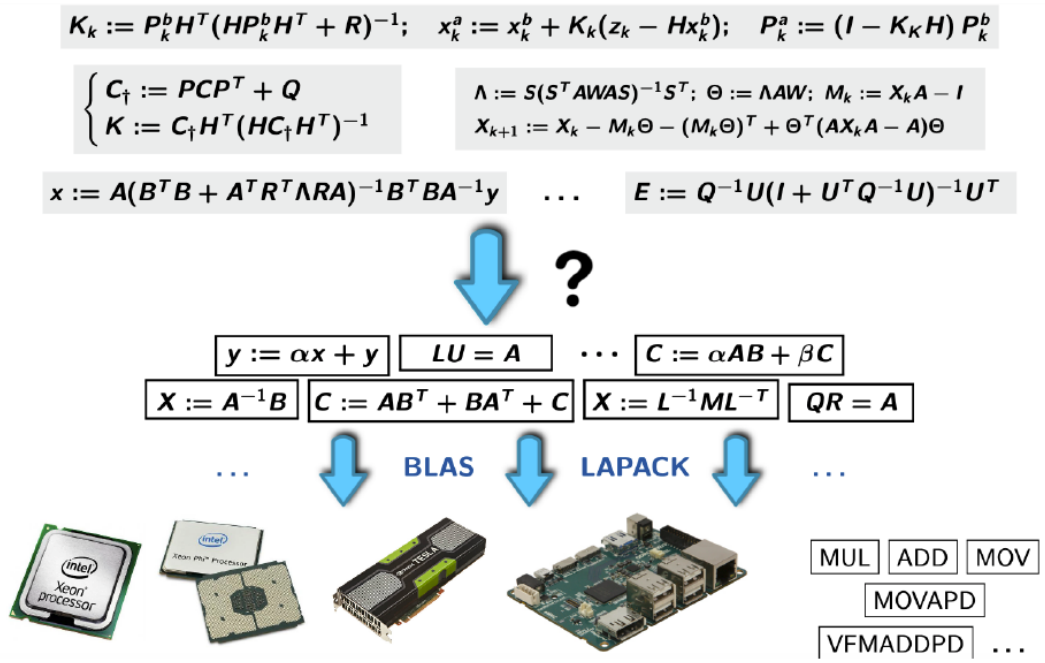
AGENDA

- About me
- The Story – Application benchmarks.
- Application Benchmarking with Directly-Follows-Graph (DFG).
- Synthesis of large amounts of I/O related system call traces.
- Final words.

THE LAMP STORY

The Linear Algebra Mapping Problem

The mapping of high level expressions to an optimized sequence of basic linear algebra operations.



THE LAMP STORY

The Linear Algebra Mapping Problem

Linear Algebra Expression

$$\mathbf{y} := H^T\mathbf{y} + (I_n - H^T H)\mathbf{x}$$

where $H \in \mathbb{R}^{n\times n}$ $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

Variant 1

$$\mathbf{y} := H^T\mathbf{y} + (I_n - H^T H)\mathbf{x}$$

Operation	Kernel	FLOPs
$\mathbf{t}_1 := H^T\mathbf{y}$	gemv	$2n^2$
$T_2 := H^T H$	gemm	$2n^3$
$T_3 := I_n - T_2$	axpy	n
$\mathbf{t}_4 := T_3\mathbf{x}$	gemv	$2n^2$
$\mathbf{y} := \mathbf{t}_1 + \mathbf{t}_4$	axpy	n

Total FLOPs $\approx 2n^3 + 4n^2 + 2n$

Exec time $\approx 0.43sec$

n=3000

executed on a single core of Intel Xeon CPU

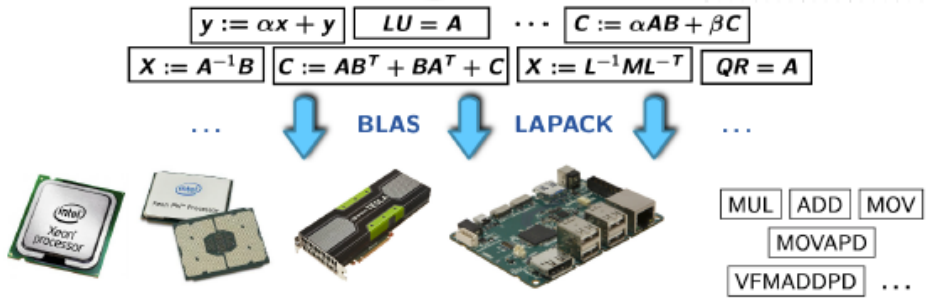
Variant 2

$$\mathbf{y} := H^T\mathbf{y} + x - H^T(H\mathbf{x})$$

Operation	Kernel	FLOPs
$\mathbf{t}_1 := H^T\mathbf{y}$	gemv	$2n^2$
$\mathbf{t}_2 := H\mathbf{x}$	gemv	$2n^2$
$\mathbf{t}_3 := H^T\mathbf{t}_2$	gemv	$2n^2$
$\mathbf{t}_4 := \mathbf{t}_1 - \mathbf{t}_3$	axpy	n
$\mathbf{y} := \mathbf{t}_4 + \mathbf{x}$	axpy	n

Total FLOPs $\approx 6n^2 + 2n$

Exec time $\approx 0.003sec$

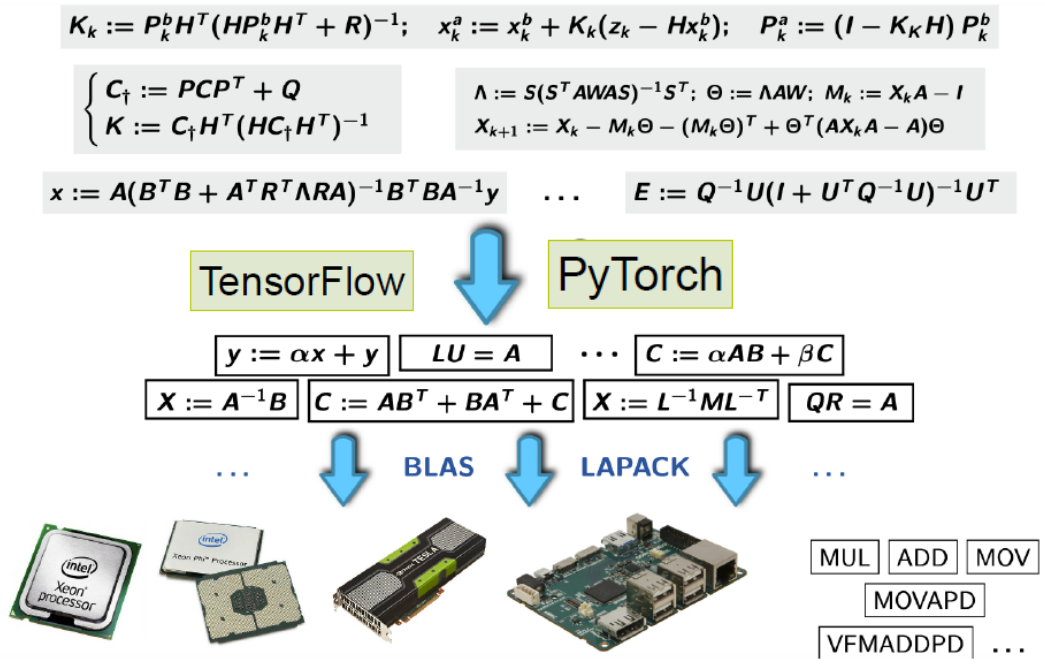


Variant 2 is orders of magnitude faster than Variant 1!

THE LAMP STORY

The Linear Algebra Mapping Problem

The mapping of high level expressions to an optimized sequence of basic linear algebra operations.



Kernels are highly optimized.

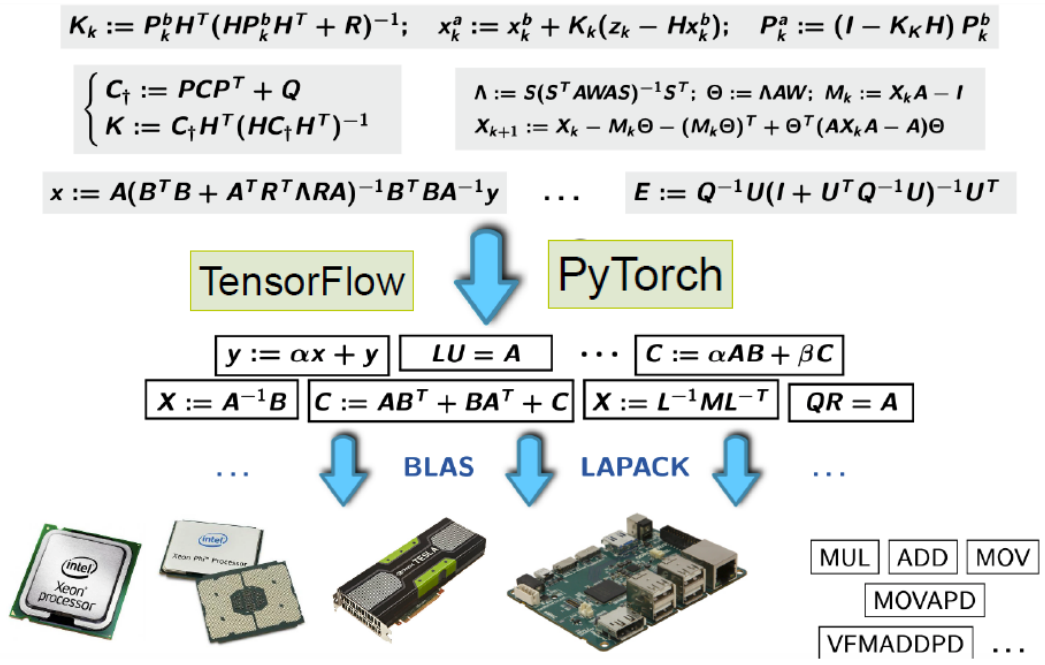
THE LAMP STORY

The Linear Algebra Mapping Problem

The mapping of high level expressions to an optimized sequence of basic linear algebra operations.

Does the mapping make best use of the available kernels?

Kernels are highly optimized.



THE LAAB STORY

The Linear Algebra Aware Benchmarks

- Developed benchmarks to evaluate the linear algebra awareness of the build of popular ML framework (*v2022*).
- Designed to run in a continuous benchmarking setup to automatically generate reports (*v2025 – yet to release*).
- *Source code:* https://github.com/HPAC/LAAB-Python

Sample CB report

LAAB-Python | LA Awareness-CPU | PyTorch/2.1.2-foss-2023a | HPC2N_x86_64

This report evaluates whether the software build performs operations equivalent to those of optimized math libraries (e.g., OpenBLAS, MKL), and whether it leverages linear algebra techniques to accelerate CPU computations. Unless stated otherwise, all benchmarks use matrices of size 3000 × 3000 and are executed on a single CPU core of AMD EPYC 7413 24-Core Processor.

1) PyTorch vs. BLAS for matrix multiplication:

	Call	time (s)
A^TB	`t(A)@B`	0.5094 ✅
"	`linalg.matmul(t(A),B)`	0.5072 ✅
Reference	`sgemm`	0.4942

2) Elimination of common sub-expression:

$E = A^TB + A^TB.$

Expr	Call	time (s)
E	`t(A)@B + t(A)@B`	0.5251 ✅
Reference	`2*(t(A)@B)`	0.5271

$E_1 = (A^TB)^T(A^TB)$ and $E_2 = (A^TB)^TA^TB.$

Expr	Call	time (s)
E_1	`t(t(A)@B)@(t(A)@B)`	1.0191 ✅
E_2	`t(t(A)@B)@t(A)@B`	1.5224 ❌
Reference	`S=t(A)@B; t(S)@S`	1.0188

THE LAAB STORY

The Linear Algebra Aware Benchmarks

- Developed benchmarks to evaluate the linear algebra awareness of the build of popular ML framework (*v2022*).
- Designed to run in a continuous benchmarking setup to automatically generate reports (*v2025 – yet to release*).
- *Source code:* https://github.com/HPAC/LAAB-Python

Sample CB report

3) Choosing the optimal order to evaluate a matrix chain:

Given m matrices of suitable sizes, the product $M = A_1 A_2 \dots A_n$ is known as a matrix chain. Because of associativity of matrix product, matrix chain can be computed in many different ways, each identified by a specific paranthesization. The paranthesization that evaluates M with the least number of floating point operations (FLOPs) is considered optimal. PyTorch provides the method `torch.linalg.multi_dot` specifically for evaluation of matrix chains.

a) **Right to left:**

The input matrix chain is $H^T H x$, where x is a vector of lenth 3000. The reference implementation, evaluating from right-to-left - i.e., $H^T(Hx)$, avoids the expensive $O(n^3)$ matrix product, and has a complexity of $O(n^2)$.

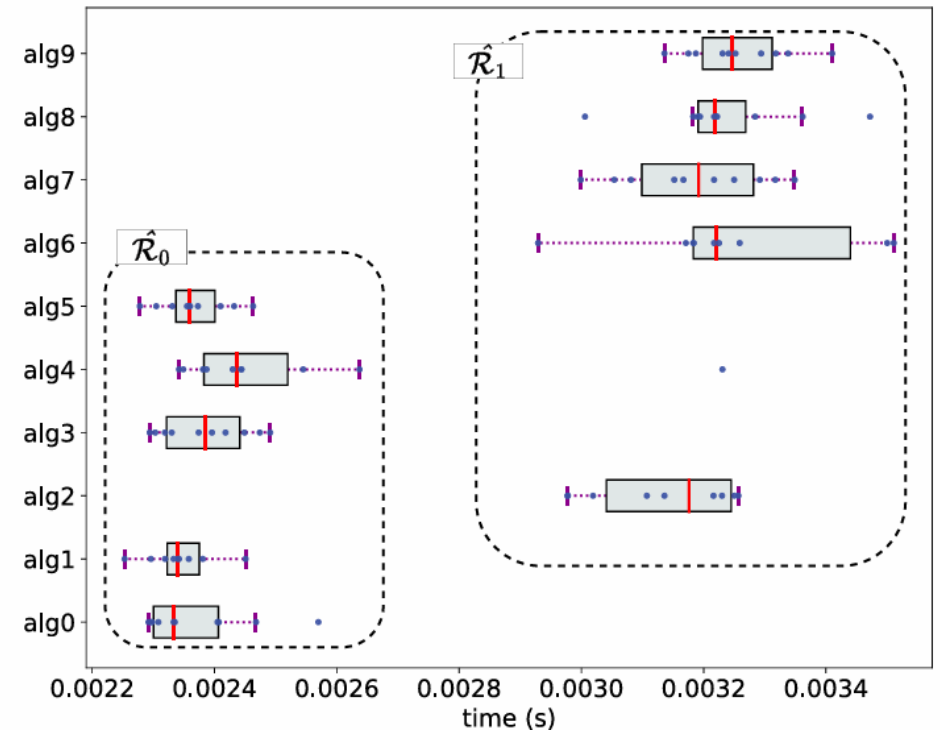
Expr	Call	time (s)
$H^T H x$	`t(H)@H@x`	0.5100 ❌
"	`linalg.multi_dot([t(H), H, x])`	0.0064 ✅
Reference	`t(H)@(H@x)`	**0.0054**

For more info, refer: A. Sankaran, N. A. Alashti, C. Psarras and P. Bientinesi, "Benchmarking the Linear Algebra Awareness of TensorFlow and PyTorch," *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Lyon, France, 2022, pp. 924-933, doi: 10.1109/IPDPSW55747.2022.00150.

BENCHMARKING WITH DFG

The algorithms use case

Identification of root causes of performance differences among algorithm variants.

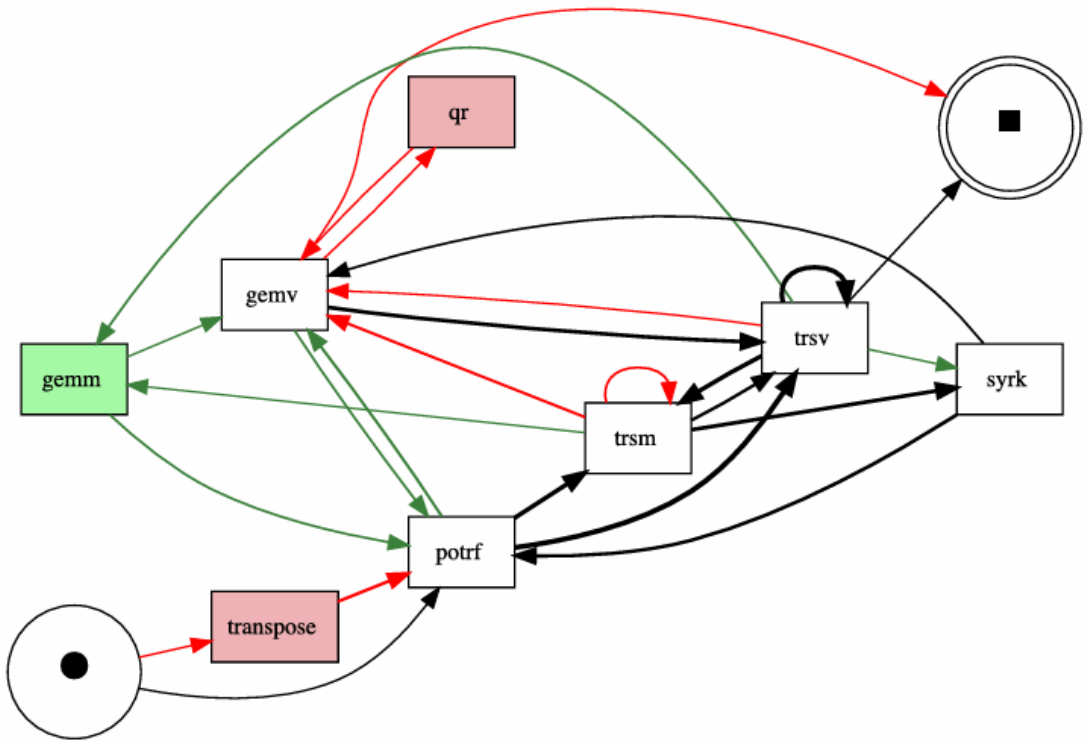


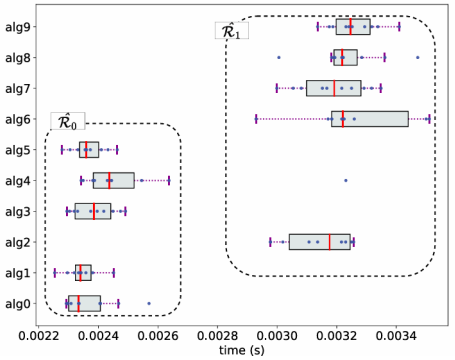
Variant	Kernel sequence
$\mathbf{alg}_0$	potrf, trsm, trsv, syrk, gemv, potrf, trsv, trsv
$\mathbf{alg}_1$	potrf, trsv, trsm, syrk, potrf, gemv, trsv, trsv
$\mathbf{alg}_2$	potrf, trsm, trsv, syrk, gemv, qr, gemv, trsv
$\mathbf{alg}_3$	potrf, trsv, trsm, gemm, potrf, gemv, trsv, trsv
$\mathbf{alg}_4$	potrf, trsm, trsv, gemm, gemv, potrf, trsv, trsv
$\mathbf{alg}_5$	potrf, trsm, trsv, gemm, potrf, gemv, trsv, trsv
$\mathbf{alg}_6$	transpose, potrf, trsm, trsv, syrk, potrf, trsv, trsm, gemv, trsv
$\mathbf{alg}_7$	transpose, potrf, trsm, syrk, potrf, trsv, trsv, trsm, gemv, trsv
$\mathbf{alg}_8$	transpose, potrf, trsm, syrk, potrf, trsm, trsm, trsv, trsv, gemv
$\mathbf{alg}_9$	transpose, potrf, trsm, syrk, potrf, trsv, trsv, trsm, trsm, gemv

Algorithmic variants of the generalized least square problem.

BENCHMARKING WITH DFG

The algorithms use case





Variant	Kernel sequence
$\mathbf{alg}_0$	potrf, trsm, trsv, syrk, gemv, potrf, trsv, trsv
$\mathbf{alg}_1$	potrf, trsv, trsm, syrk, potrf, gemv, trsv, trsv
$\mathbf{alg}_2$	potrf, trsm, trsv, syrk, gemv, qr, gemv, trsv
$\mathbf{alg}_3$	potrf, trsv, trsm, gemm, potrf, gemv, trsv, trsv
$\mathbf{alg}_4$	potrf, trsm, trsv, gemm, gemv, potrf, trsv, trsv
$\mathbf{alg}_5$	potrf, trsm, trsv, gemm, potrf, gemv, trsv, trsv
$\mathbf{alg}_6$	transpose, potrf, trsm, trsv, syrk, potrf, trsv, trsm, gemv, trsv
$\mathbf{alg}_7$	transpose, potrf, trsm, syrk, potrf, trsv, trsv, trsm, gemv, trsv
$\mathbf{alg}_8$	transpose, potrf, trsm, syrk, potrf, trsm, trsm, trsv, trsv, gemv
$\mathbf{alg}_9$	transpose, potrf, trsm, syrk, potrf, trsv, trsv, trsm, trsm, gemv

Ref: Sankaran, A., Karlsson, L. & Bientinesi, P. Ranking with ties based on noisy performance data. *Int J Data Sci Anal* (2025). https://doi.org/10.1007/s41060-025-00722-1

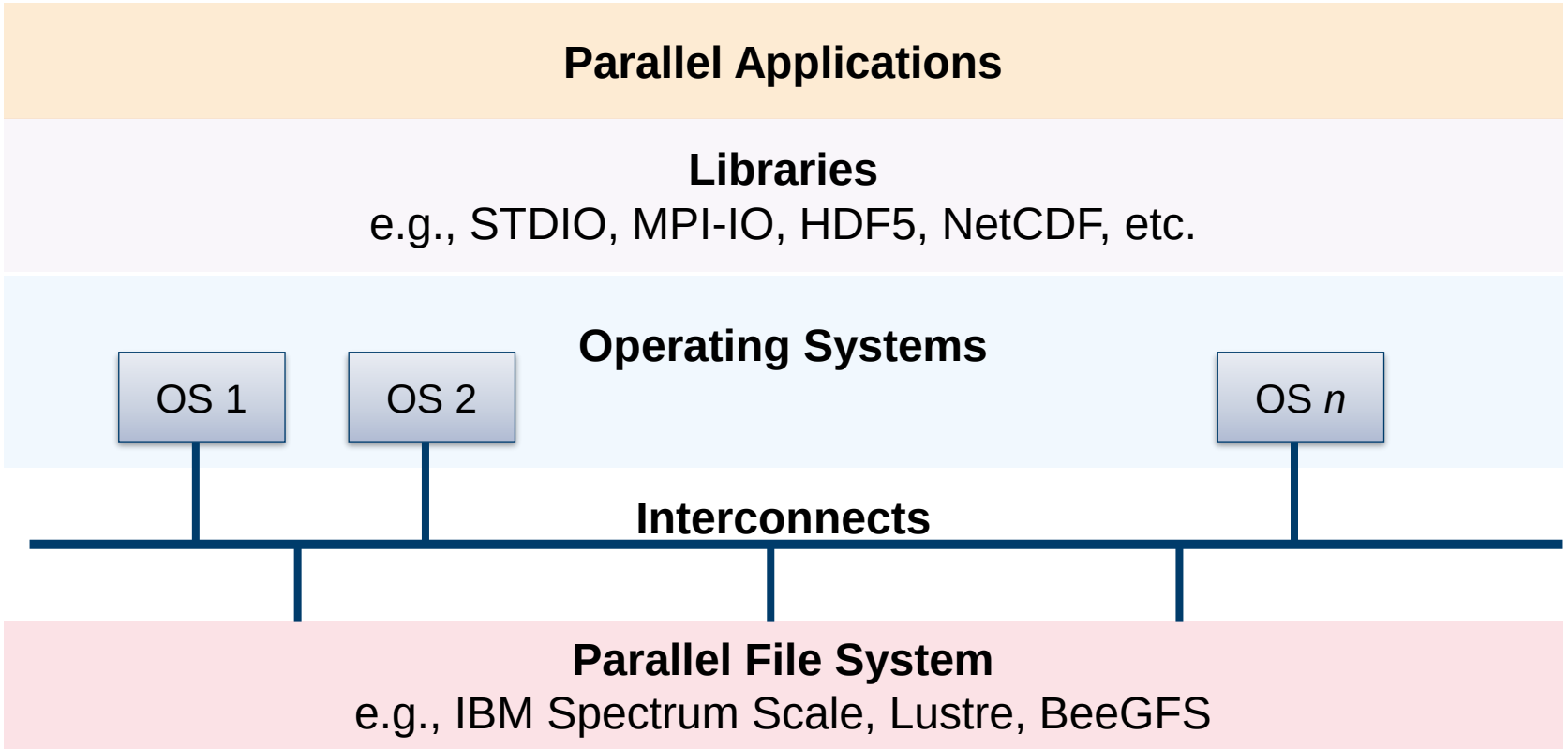
AND NOW,

The picture is incomplete without I/O information.

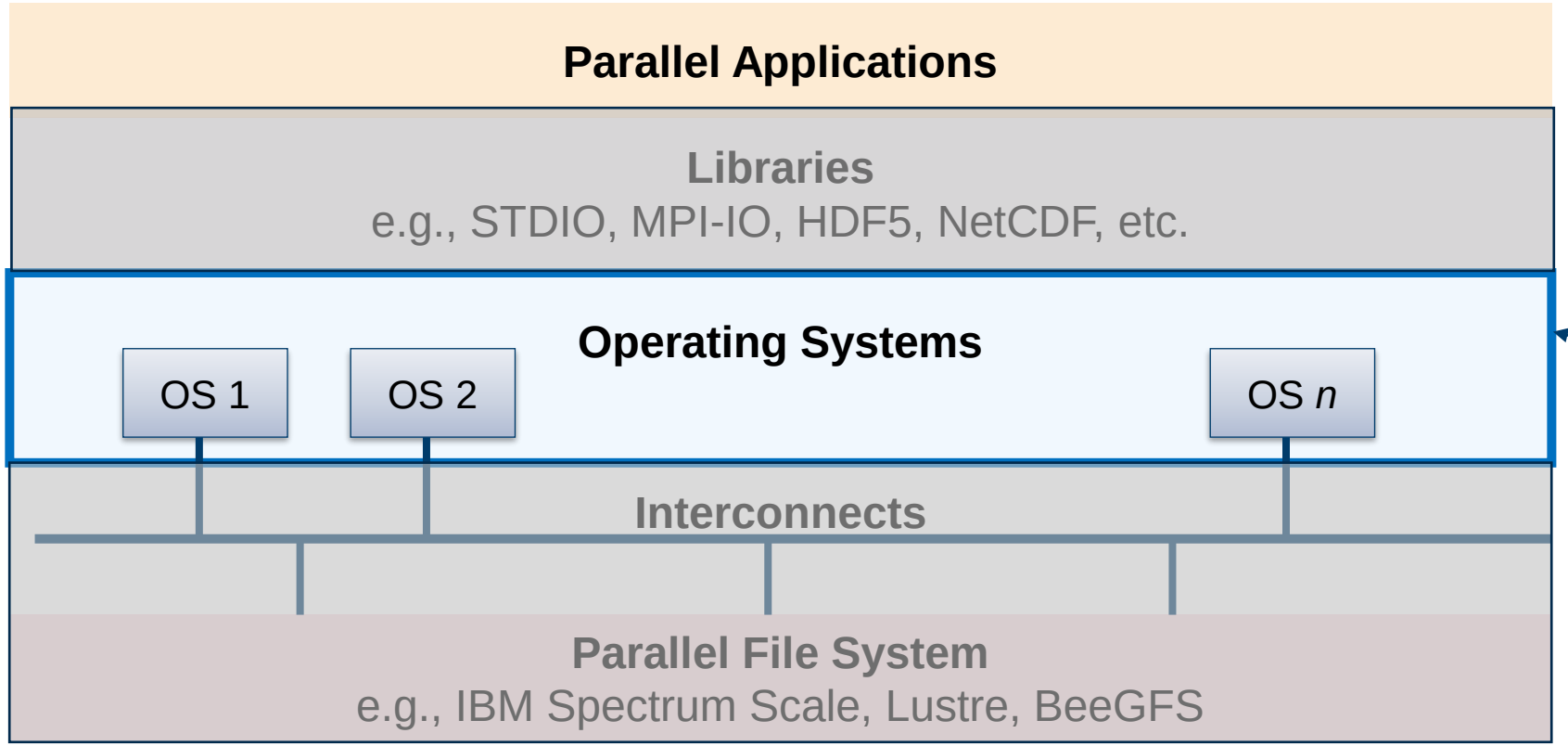
The Problem:

- How to include I/O in LAAB reports?
- Investigate the use of DFG method to ***synthesize*** the large amounts of I/O performance data.

WHICH I/O DATA TO USE?



WHICH I/O DATA TO USE?



TRACING WITH STRACE

Basic Usage:

```
strace [COMMAND]
```

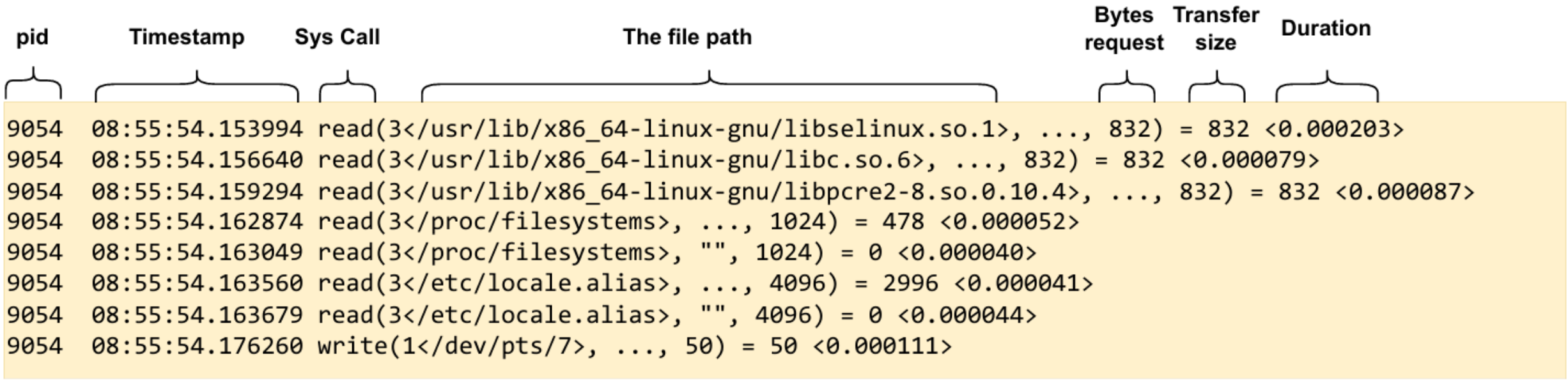
Example:

```
$ strace ls
execve("/usr/bin/ls", ["ls"], 0x7ffd9ef46ac0 /* 36 vars */) = 0
brk(NULL)                               = 0x56490ef01000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc49b7b440) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9f2c1d8000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=38711, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 38711, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9f2c1ce000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
```

TRACING WITH STRACE

Example:

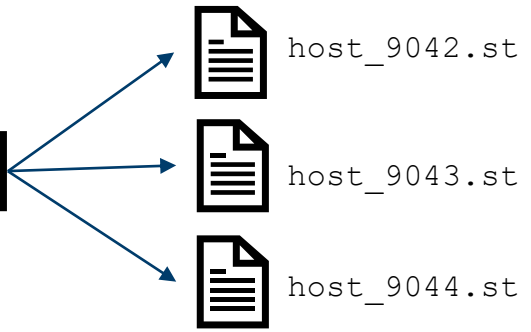
```
strace -f -tt -T -y -e read,write  ls
```



TRACING WITH STRACE

Example:

```
srun -n 3 strace -f -tt -T -y -e read,write -o $hostname_$$.st ls
```



pid | Timestamp | Sys Call | The file path | Bytes request | Transfer size | Duration

```
9054  08:55:54.153994 read(3</usr/lib/x86_64-linux-gnu/libselinux.so.1>, ..., 832) = 832 <0.000203>
9054  08:55:54.156640 read(3</usr/lib/x86_64-linux-gnu/libc.so.6>, ..., 832) = 832 <0.000079>
9054  08:55:54.159294 read(3</usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.10.4>, ..., 832) = 832 <0.000087>
9054  08:55:54.162874 read(3</proc/filesystems>, ..., 1024) = 478 <0.000052>
9054  08:55:54.163049 read(3</proc/filesystems>, "", 1024) = 0 <0.000040>
9054  08:55:54.163560 read(3</etc/locale.alias>, ..., 4096) = 2996 <0.000041>
9054  08:55:54.163679 read(3</etc/locale.alias>, "", 4096) = 0 <0.000044>
9054  08:55:54.176260 write(1</dev/pts/7>, ..., 50) = 50 <0.000111>
```

Trace file: host_9042.st

THE CHALLENGE

	pid	call	start	duration	bytes	fs	case	end
10	22085	read	1900-01-01 18:54:16.207116	0.000015	832	/p/software/fs/jusuf/stages/2024/software/HDF5...	st.jsfc134.22051.log	1900-01-01 18:54:16.207131
33	22085	read	1900-01-01 18:54:16.211619	0.000017	832	/p/software/fs/jusuf/stages/2024/software/psmp...	st.jsfc134.22051.log	1900-01-01 18:54:16.211636
221	22085	read	1900-01-01 18:54:16.246555	0.000008	832	/usr/lib64/libc.so.6	st.jsfc134.22051.log	1900-01-01 18:54:16.246563
231	22085	read	1900-01-01 18:54:16.247709	0.000015	832	/p/software/fs/jusuf/stages/2024/software/IME/...	st.jsfc134.22051.log	1900-01-01 18:54:16.247724
235	22085	read	1900-01-01 18:54:16.248466	0.000016	832	/p/software/fs/jusuf/stages/2024/software/Szip...	st.jsfc134.22051.log	1900-01-01 18:54:16.248482

Challenge: How do you extract useful information from large amounts of information in the system call traces?

THE SYNTHESIS METHOD

Idea:

- **Map each row to** a string that helps answer your question. We call this string "**Activity**".
- **Apply grouping** based on activities and **compute statistics**.
- **Identify the directly-follows relation** between the activities to build a **Directly-Follows-Graph (DFG).**

pid	call	start	duration	bytes	fs
22085	read	1900-01-01 18:54:16.207116	0.000015	832	/p/software/fs/jusuf/stages/2024/software/HDF5...
22085	read	1900-01-01 18:54:16.211619	0.000017	832	/p/software/fs/jusuf/stages/2024/software/psmp...
22085	read	1900-01-01 18:54:16.246555	0.000008	832	/usr/lib64/libc.so.6

Activity

→ read+/p/software

→ read+/p/software

→ read+/usr/lib64

THE SYNTHESIS METHOD

pid	call	start	duration	bytes	fs
22085	read	1900-01-01 18:54:16.207116	0.000015	832	/p/software/fs/jusuf/stages/2024/software/HDF5...
22085	read	1900-01-01 18:54:16.211619	0.000017	832	/p/software/fs/jusuf/stages/2024/software/psmp...
22085	read	1900-01-01 18:54:16.246555	0.000008	832	/usr/lib64/libc.so.6

Activity

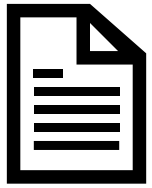
- read+/p/software
- read+/p/software
- read+/usr/lib64

- This data is can be formalized as an **event-log** in **Process Mining**.
- Process mining introduces **scalable techniques to translate event logs into** different types of dependency graphs, including **Directly-Follows Graph**.
- Ref: W. M. P. Van Der Aalst, “Foundations of process discovery,” in Process Mining Handbook, DOI: https://doi.org/10.1007/978-3-031-08848-3_2

THE SYNTHESIS METHOD

P0.strace.log

Activity
read+$SOFTWARE
read+$SOFTWARE
write+$PROJECT



P1.strace.log

Activity
read+$SOFTWARE
write+$PROJECT
write+$SCRATCH

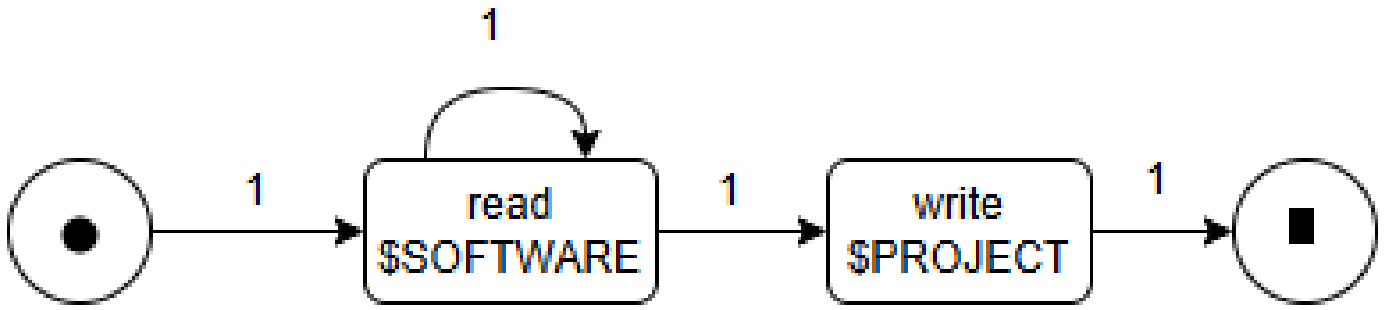
THE SYNTHESIS METHOD

P0

Activity
read+$SOFTWARE
read+$SOFTWARE
write+$PROJECT

P1

Activity
read+$SOFTWARE
read+$PROJECT
write+$PROJECT



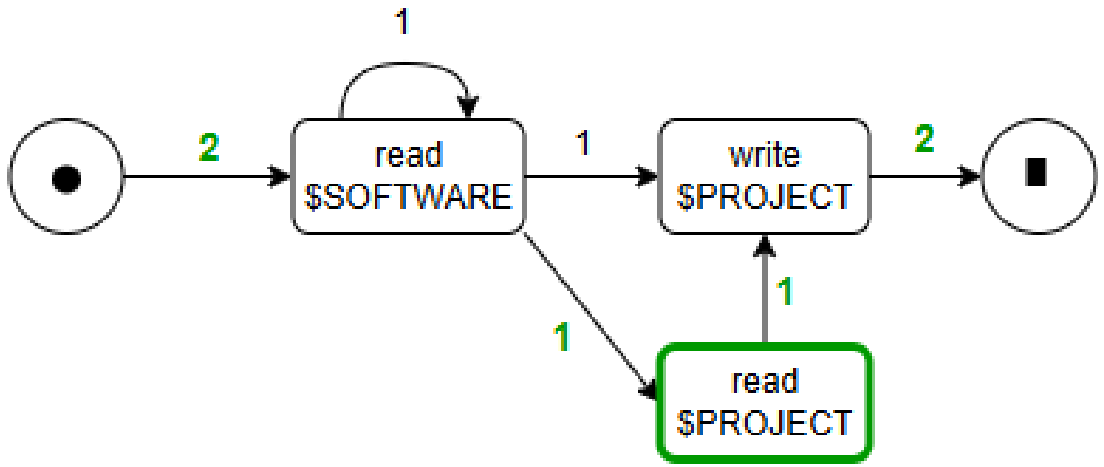
THE SYNTHESIS METHOD

P0

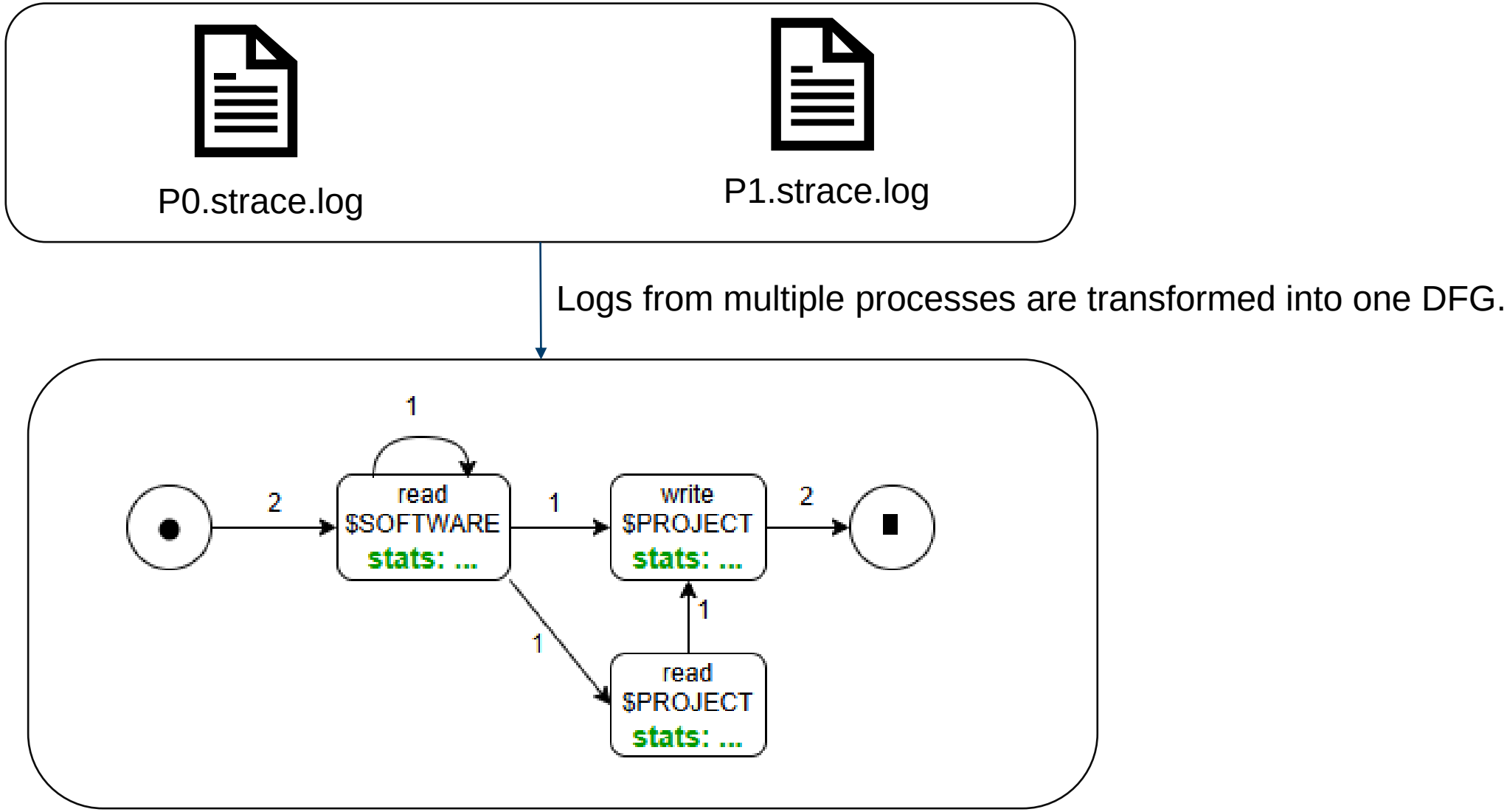
Activity
read+$SOFTWARE
read+$SOFTWARE
write+$PROJECT

P1

Activity
read+$SOFTWARE
read+$PROJECT
write+$PROJECT

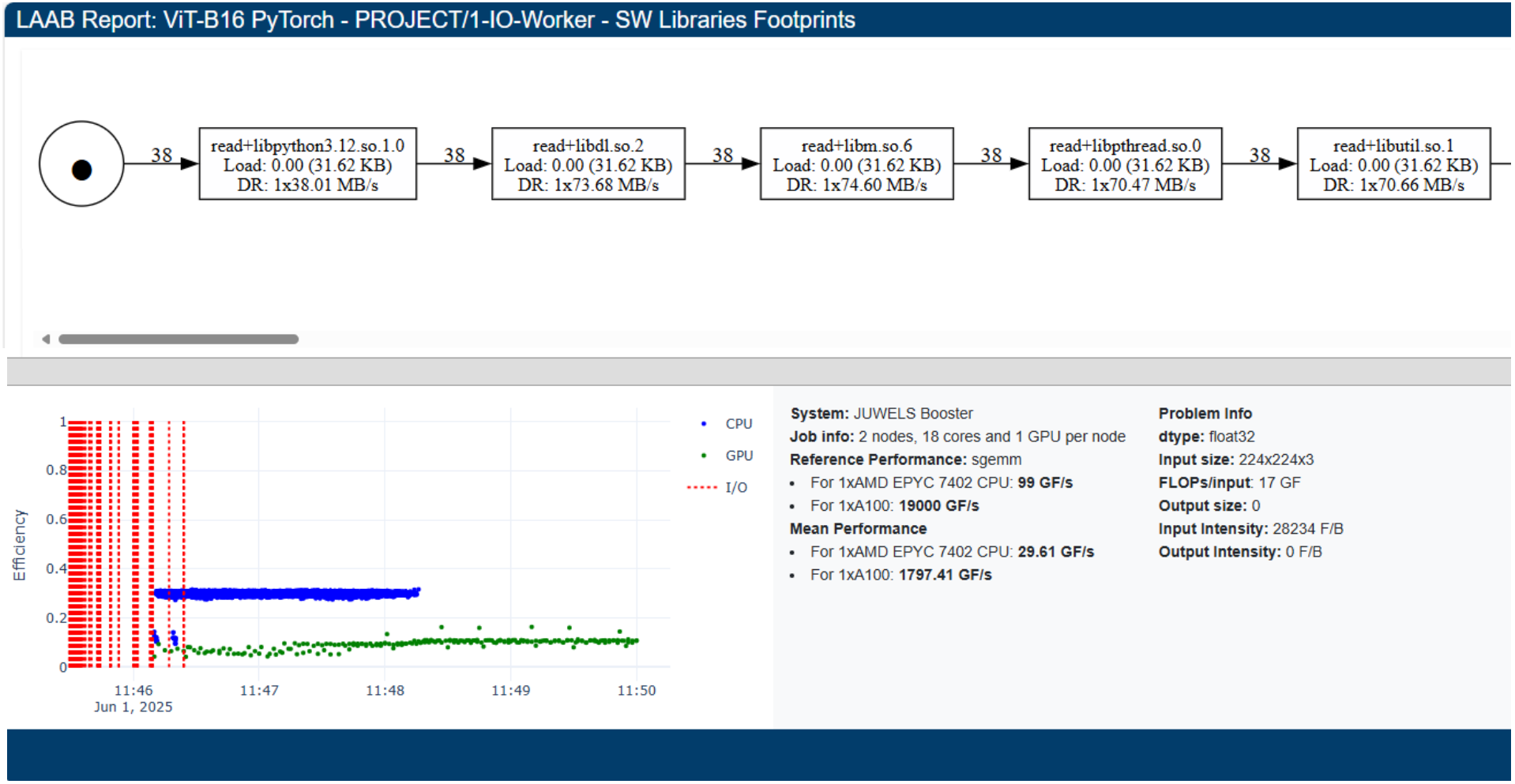


THE SYNTHESIS METHOD



DEMO

SNAPSHOT OF LAAB REPORT WITH I/O



THE IMPLEMENTATION: STRACE INSPECTOR

- One log file is generated per node used.
- The parsing of log files and mapping to activities are done in parallel (*in v1.1.0*).
- The construction of DFG requires one pass through the activity log - O(n), where n is the number of lines in the filtered event log after mapping.
- The complexity of rendering the DFG is O(m2), where m is the number activities. For all intents and purposes, m should be small.

Source code: https://gitlab.jsc.fz-juelich.de/st-inspector/st_inspector

Ref: A. Sankaran, I. Zhukov, W. Frings and P. Bientinesi, "Inspection of I/O Operations from System Call Traces using Directly-Follows-Graph," *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Atlanta, GA, USA, 2024, pp. 1562-1575, doi: 10.1109/SCW63240.2024.00196

Acknowledgements

Thank you for you attention.

- This research was financially supported by the Juelich Supercomputing Center at Forschungszentrum Juelich and the BMBF project 01-1H1-6013 AP6 NRW Anwenderunterstutzung SiVeGCS.
- Additionally, supervision support from RWTH Aachen University through the DFG project IRTG-2379 is gratefully acknowledged.
- Compute time on JUWELS Booster at JSC and Kebnekaise at HPC2N (Sweden) is gratefully acknowledged.