

Distributed Summary Synthesis Multi-processing

Bringing Software Verification to the Cloud

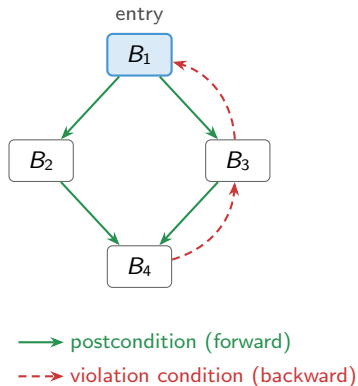
Akshay Warriar, Thomas Lemberger, Matthias Kettl, Dirk Beyer

CPAchecker Workshop, ASE 2026

16.10.2026

Recap: Distributed Summary Synthesis

- Decompose the program into a block graph
- Spawn a worker per block
- Workers exchange conditions:
 - **postconditions** to successors
 - **violation conditions** to predecessors
- A violation condition reaches the root block → **FALSE**
- When all workers have finished → **TRUE**



Why DSS Multi-processing?

Limitations of DSS as implemented in CPAchecker:

1. Does not have the ability to distribute work load across multiple compute nodes

Why DSS Multi-processing?

Limitations of DSS as implemented in CPAchecker:

1. Does not have the ability to distribute work load across multiple compute nodes
2. Spawning one thread per block is inflexible; Most threads are idle.

Why DSS Multi-processing?

Limitations of DSS as implemented in CPAchecker:

1. Does not have the ability to distribute work load across multiple compute nodes
2. Spawning one thread per block is inflexible; Most threads are idle.
3. Multi-threaded work fully implemented in CPAchecker → difficult to understand, all tooling must be built onto CPAchecker directly

Why DSS Multi-processing?

Limitations of DSS as implemented in CPAchecker:

1. Does not have the ability to distribute work load across multiple compute nodes
2. Spawning one thread per block is inflexible; Most threads are idle.
3. Multi-threaded work fully implemented in CPAchecker → difficult to understand, all tooling must be built onto CPAchecker directly
4. No fault tolerance: a single crash loses all analysis progress

Why DSS Multi-processing?

Limitations of DSS as implemented in CPAchecker:

1. Does not have the ability to distribute work load across multiple compute nodes
2. Spawning one thread per block is inflexible; Most threads are idle.
3. Multi-threaded work fully implemented in CPAchecker → difficult to understand, all tooling must be built onto CPAchecker directly
4. No fault tolerance: a single crash loses all analysis progress
5. Message format is tied to CPAchecker internals

DSS as a set of cloud-native microservices

- **Scalable:** ability to distribute work across multiple nodes

Goals

DSS as a set of cloud-native microservices

- **Scalable:** ability to distribute work across multiple nodes
- **Load-balancing:** any worker should be able to analyze any block

DSS as a set of cloud-native microservices

- **Scalable:** ability to distribute work across multiple nodes
- **Load-balancing:** any worker should be able to analyze any block
- **Robust:** survive worker crashes and network failures

DSS as a set of cloud-native microservices

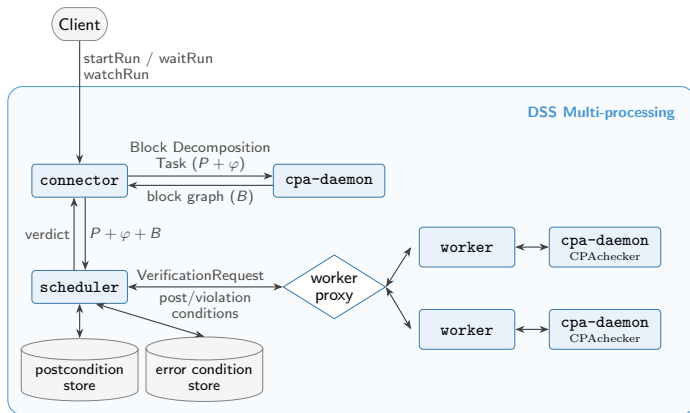
- **Scalable:** ability to distribute work across multiple nodes
- **Load-balancing:** any worker should be able to analyze any block
- **Robust:** survive worker crashes and network failures
- **Observable:** every message can be persistently replayed and every analysis is traceable

DSS as a set of cloud-native microservices

- **Scalable:** ability to distribute work across multiple nodes
- **Load-balancing:** any worker should be able to analyze any block
- **Robust:** survive worker crashes and network failures
- **Observable:** every message can be persistently replayed and every analysis is traceable
- **Verifier-agnostic:** interfaces in protobuf; CPAchecker is today's backend

- `cpa-daemon` service [ATVA'24]: CPAchecker as a long-running gRPC service.
 - Does not spawn a new JVM process for every analysis
 - Enables us to cache static objects: CFA, specification, block graph, etc.
- `worker` service: light-weight service that forwards requests to `cpa-daemon`
- Any worker should be able to analyze any block; workers can be added, removed or crash without losing progress

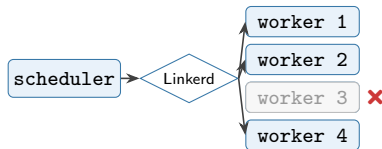
Architecture and Analysis Workflow



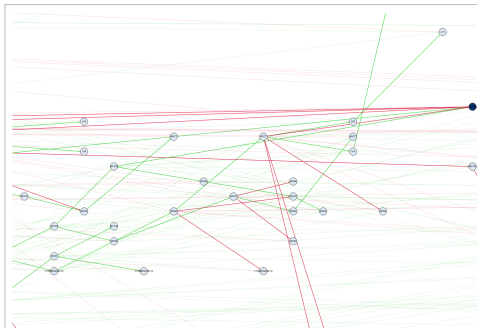
- connector receives program P and specification φ and decomposes P into the block graph B
- scheduler sends a request for every block, stores the returned conditions and schedules the neighbours
- worker analyzes one block on its cpa-daemon and returns post/violation conditions
- Repeat until a verdict is reached

Infrastructure Features

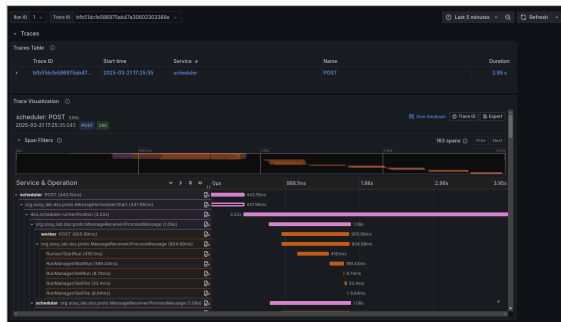
- **Kubernetes** places, restarts and scales the services; fixed CPU and memory limits per container make measurements reproducible
- **Linkerd** balances gRPC per **request**, so requests spread over all worker pods
- **Failures**: if a cpa-daemon crashes, its worker leaves the pool until it recovers; the scheduler retries with exponential backoff and marks the run **HEALTH_DEGRADED**
- **Observability**: every gRPC call is traced (OpenTelemetry); CPAchecker statistics per request are collected automatically without manual instrumentation



Visualization and Dashboard



Visualization: messages between blocks
(**postconditions**, **violation conditions**)



Dashboard: traces of a run across all services

Research Question

How much overhead does the distributed architecture of DSS Multi-processing add, compared to DSS in a single local process?

Evaluation: Experimental Setup

- **Benchmark:** 1008 tasks from sv-benchmarks, a subset of tasks that DSS was historically known to solve
- **Time limit:** 300 s walltime per task for both tools
- **Environment:** Both executed on AMD EPYC 7713 64-Core Processor. DSSM used minikube to start a local Kubernetes cluster. DSSI ran using benchexec.
- All DSSM services are restarted before every task. cputime and memory queried after each task from container's cgroup stats.

Results: Verdicts

Verdict confusion matrix

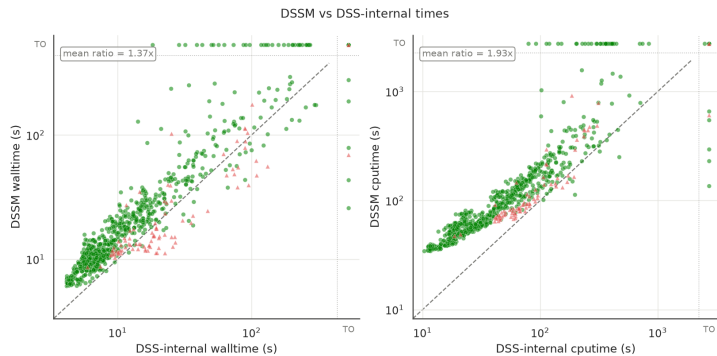
DSSM verdict	Correct T	813	0	0	0	11
	Correct F	0	91	0	0	5
	Wrong T	0	0	0	0	0
	Wrong F	0	0	0	0	4
	Unknown	34	0	0	0	50
		Correct T	Correct F	Wrong T	Wrong F	Unknown
		DSS-internal verdict				

- **Correct verdicts**

- DSSM: 920 (824 true, 96 false)
- DSSI: 938 (847 true, 91 false)

- Achieves the same verdict on 954 of 1008 tasks
- DSSM solves 16 tasks that DSSI does not; DSSI solves 34 that DSSM does not
- 4 false alarms from DSSM also appear in DSSI, but its witness export refutes the violations

Results: Time



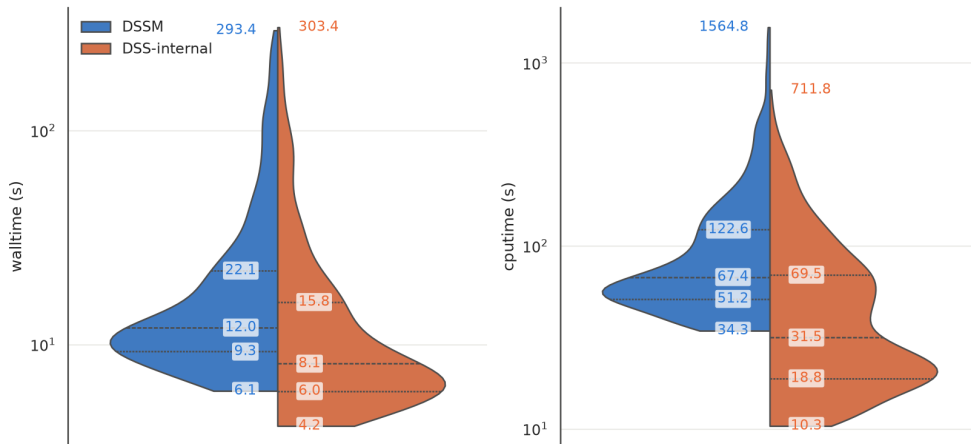
On average:

DSSM walltime **1.37×**

DSSM CPU time **1.93×**

Results: Time Distribution

Wall-clock time and CPU time distributions, both-correct tasks



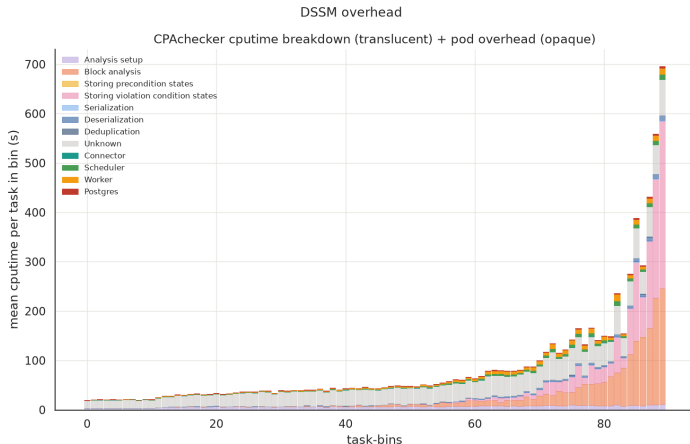
Results: Where Does the Overhead Come From?

Mean CPU time per operation (s)	DSS-internal	DSSM	Change
Analysis setup	2.89	6.63	+129.2%
Block analysis	16.69	19.54	+17.1%
Loading precondition states into memory	0.08	0.04	-55.7%
Loading violation condition states into memory	6.51	17.54	+169.3%

Statistics as reported by CPAchecker

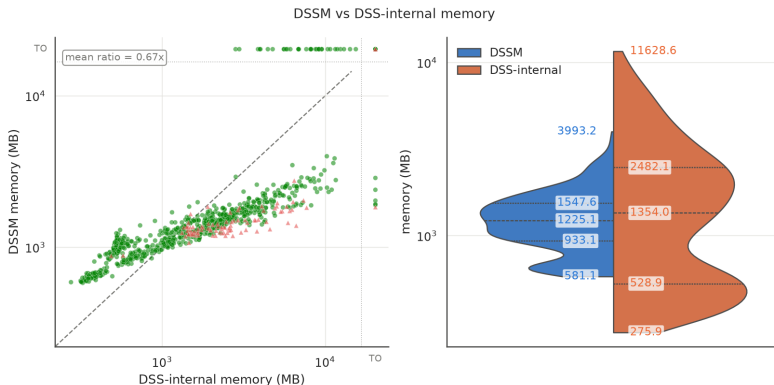
DSSM's single workers have to load the **known conditions** in every request. Scales with: **predecessors/successors, single-block analyses**

Results: DSSM Overhead



connector,
scheduler, worker
and postgres do **neg-
ligible** work compared
to the cpa-daemon

Results: Memory

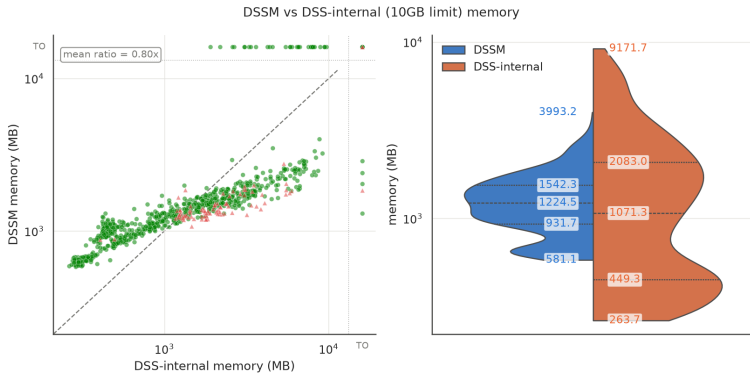


DSSM needs on average **0.67** $\times$ the memory of DSSI

Does CPAchecker allocate more memory simply because its limit is higher?

DSSI: 16 GB vs. cpa-daemon in DSSM: 10 Gi

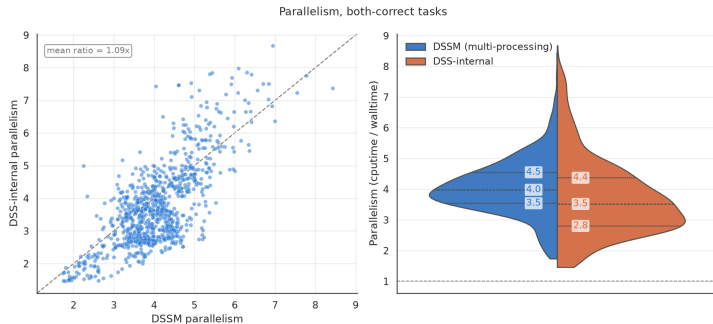
Results: Memory with a 10 GB CPAchecker Limit



DSSM needs on average **0.80** $\times$ the memory of DSSI

- DSSM creates and cleans up CPAchecker runs per block analysis
- Some DSSI analyses can spawn large number of threads ($>$ than 100 blocks)

Results: Parallelism



$$parallelism_{DSSM} = \frac{\text{cpa-daemon CPU time}}{\text{walltime}}$$

$$parallelism_{DSSI} = \frac{\text{CPAchecker CPU time}}{\text{walltime}}$$

Scaling beyond the current setup, with more cores or more workers, is **not beneficial** for the chosen benchmark set

Conclusion

- We provide an infrastructure that runs DSS as stateless microservices on Kubernetes, that is scalable with load balancing, fault tolerance and tracing
- Same verdict as DSSI on 954 of 1008 tasks
- Overhead of **$1.37\times$** walltime and **$1.93\times$** CPU time while the memory peak is only **$0.67\text{--}0.80\times$** that of DSSI

- Benchmark tasks with more opportunities for parallelism, or test different block decomposition strategies
- Incremental verification
- Generalize multi-processing to more abstract domains and tools

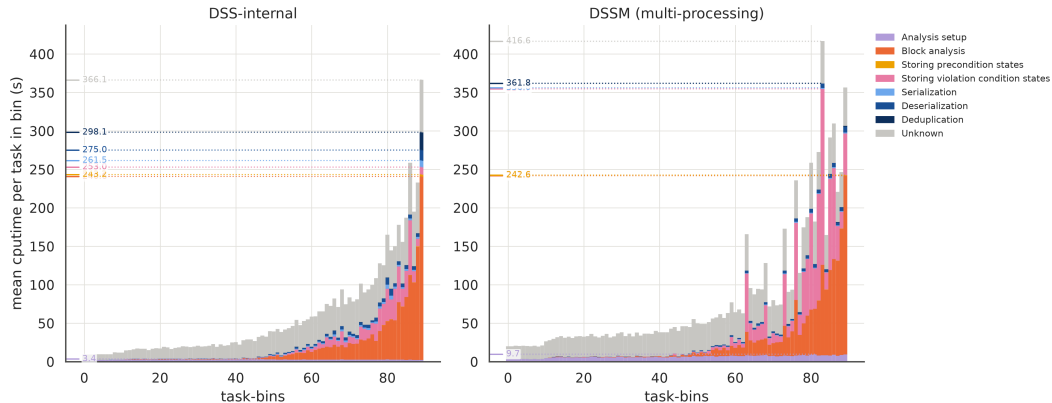
Resources: 16 CPU cores and 16 GB of memory for each tool

DSSM Kubernetes Resource Distribution

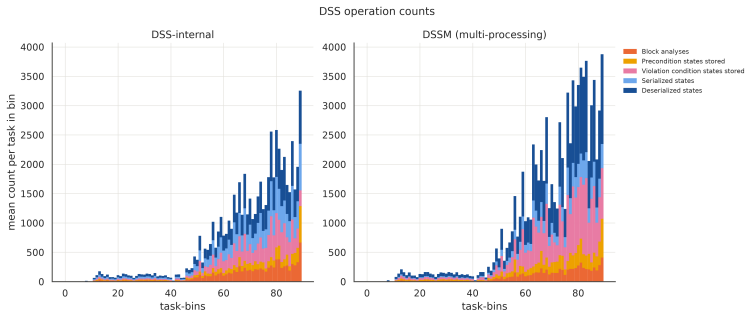
Deployment	Container	CPU	Memory
worker	worker	2	2 Gi
	cpa-daemon	9	10 Gi
scheduler	scheduler	1	1 Gi
connector	connector	1	1 Gi
	cpa-daemon	2	1 Gi
postgres	postgres	1	1 Gi

Backup: CPU Time Breakdown

DSS CPAchecker cputime breakdown



Backup: Operation Counts



Mean count per operation	DSS-internal	DSSM	Change
Block analyses	100.53	82.90	−17.5%
Precondition states loaded into memory	58.17	93.13	+60.1%
Violation condition states loaded into memory	142.58	282.28	+98.0%
Serialized states	158.22	113.63	−28.2%
Deserialized states	202.73	375.41	+85.2%

Statistics as reported by CPAchecker