Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern

WICSA/CompArch 2016

Christian Wulf,
Christian Claus Wiechmann, and
Wilhelm Hasselbring

07.04.2016

Software Engineering Group
Kiel University, Germany

The Pipe-and-Filter (P&F) Style

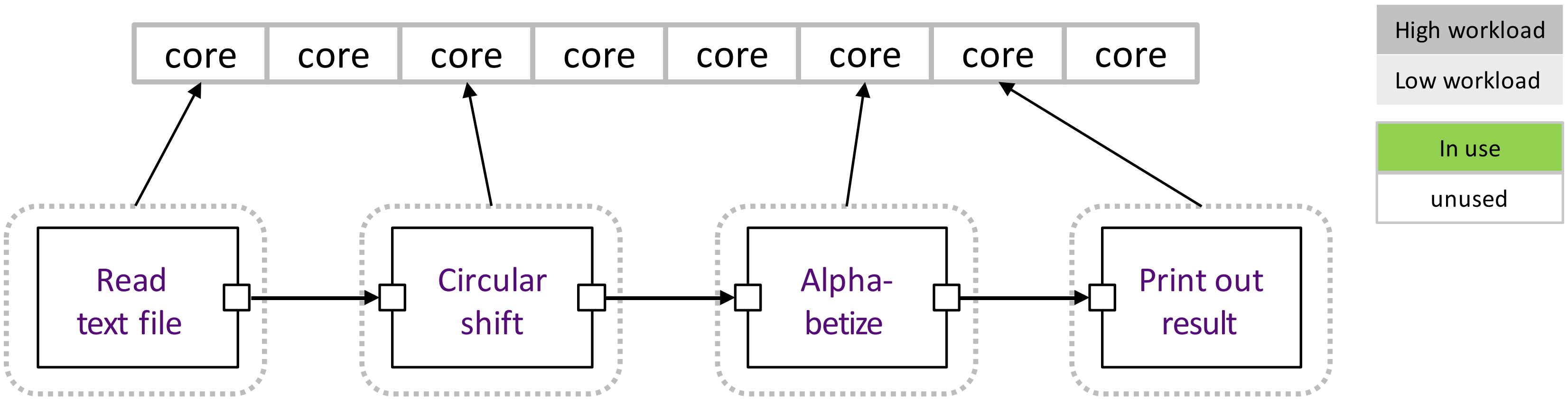
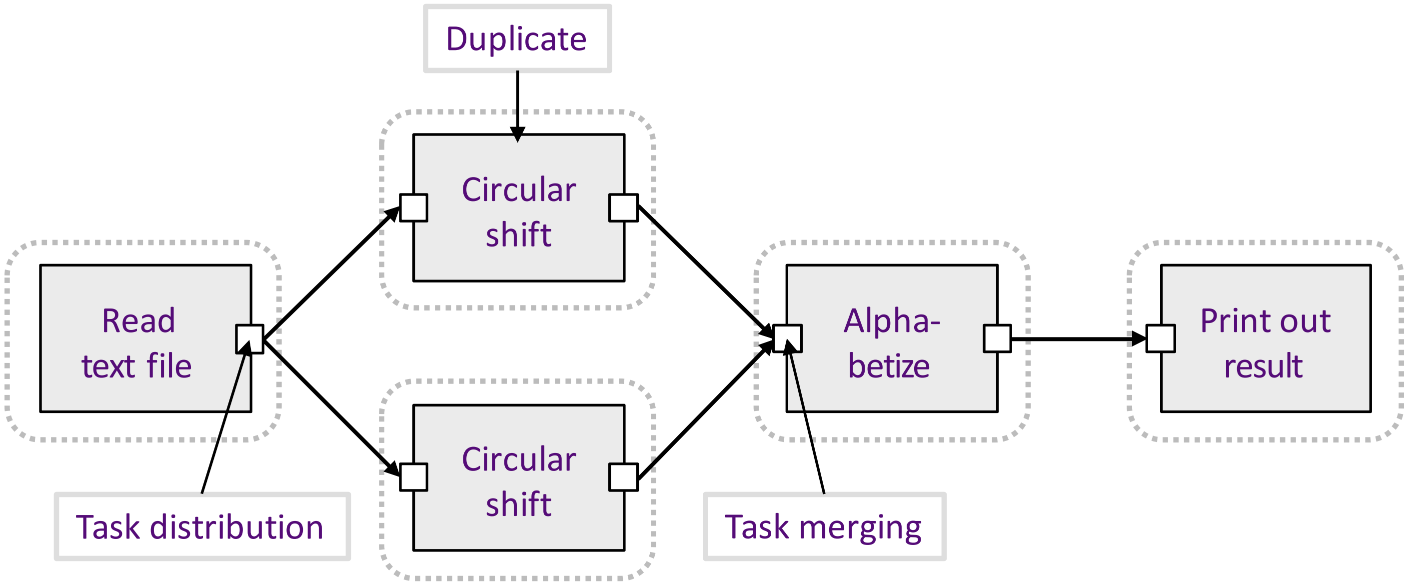


Figure 1: An example pipeline: Parnas' Keyword In Context program [Parnas1972] as P&F implementation [Rayside2006]

- Challenge: how to leverage contemporary systems for a high throughput?
- One simple approach is to execute each filter concurrently.
- Less effective for unevenly distributed workloads and for too many processing units.

A Possible Solution with 2 Stages



Advantages:
- Schedules the workload among the stages
- Scales statically with the number of processing units

Resulting challenges:
- How and where to distribute efficiently?
- How to merge efficiently?
- How to duplicate the filter?
- Computation cost >> communication cost
- Unbalanced workloads

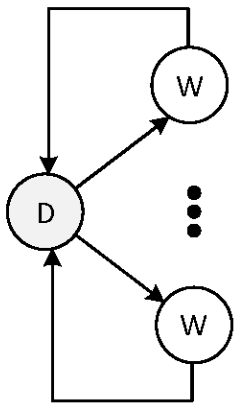
Outline

Christian-Albrechts-Universität zu Kiel

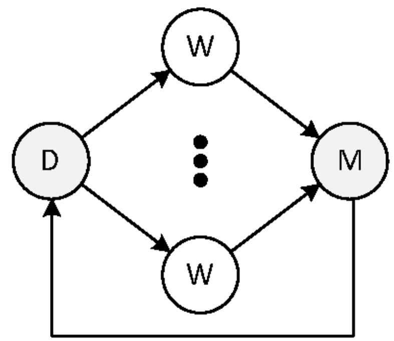
- Motivation
- The Task Farm Parallelization Pattern
- Our Approach
- Evaluation
- Related Work
- Conclusions

The Task Farm Parallelization Pattern

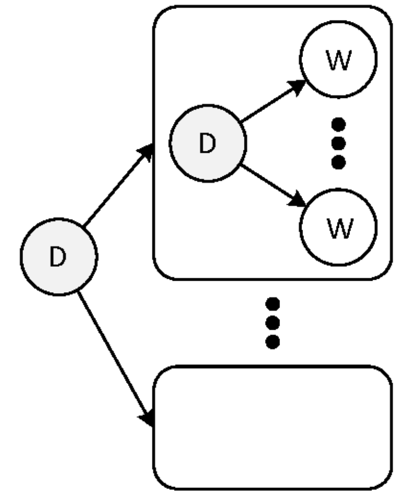
[Cole1991, Aldinucci+1999]



Farm as master/worker



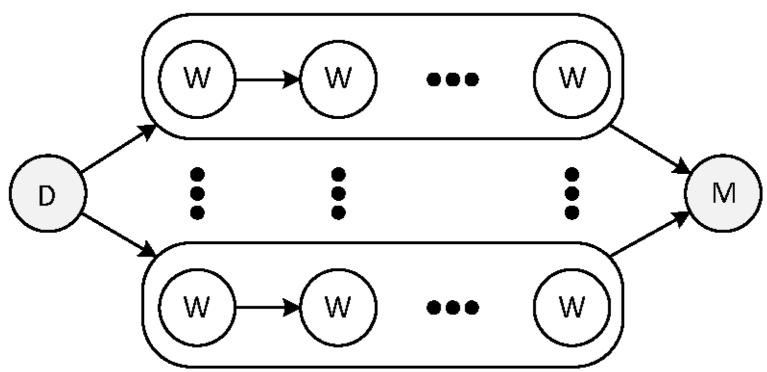
Farm with a merger and a feedback loop



Hierarchical farm composed of farms

Problem

Perform a function on each task of a given stream of tasks



Hierarchical farm composed of pipelines

Solution

Use a task distributor and a task merger with active duplicated worker filters

Outline

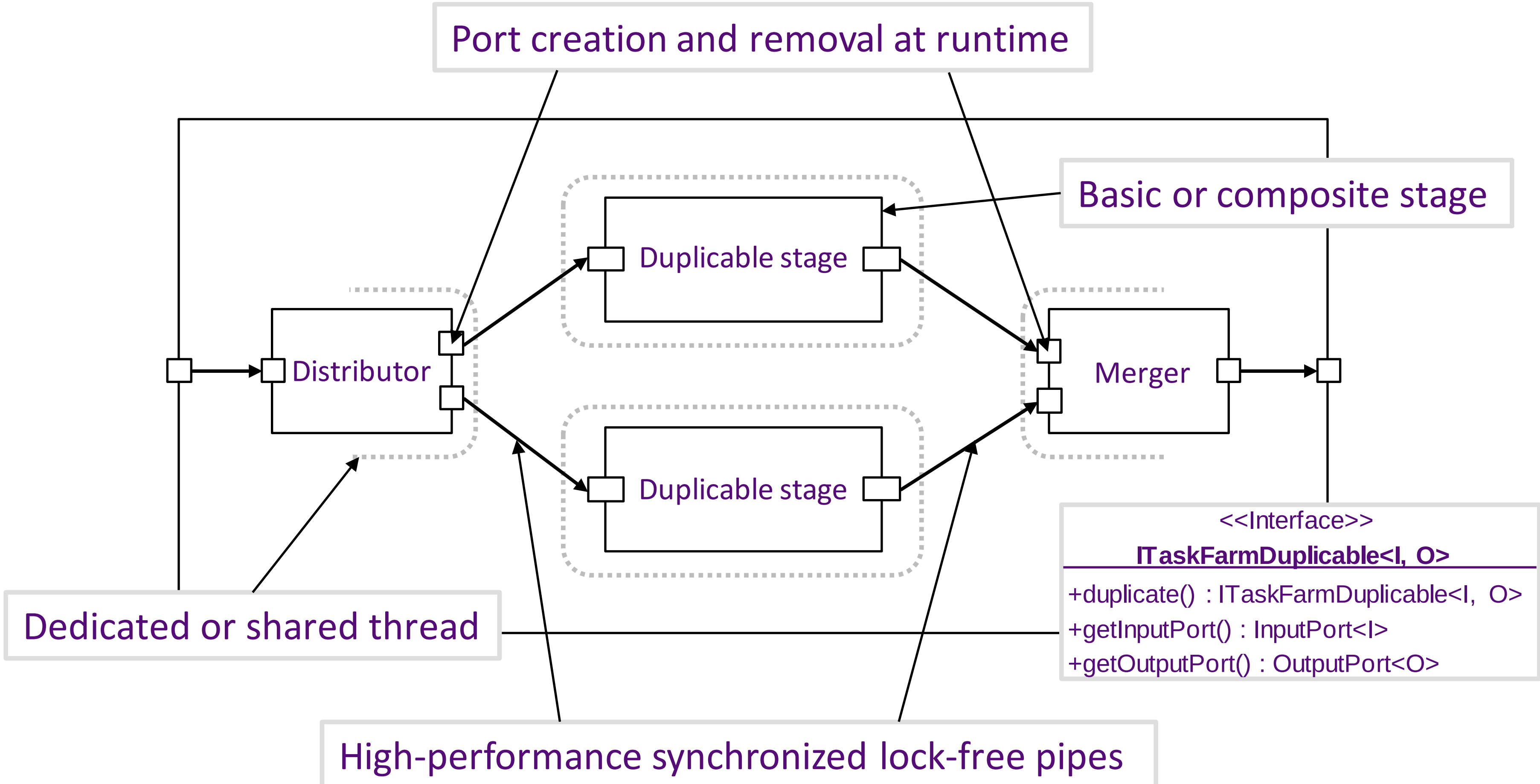
Christian-Albrechts-Universität zu Kiel

- Motivation
- The Task Farm Parallelization Pattern
- **Our Approach**
- Evaluation
- Related Work
- Conclusions

Our Task Farm Stage (TFS)

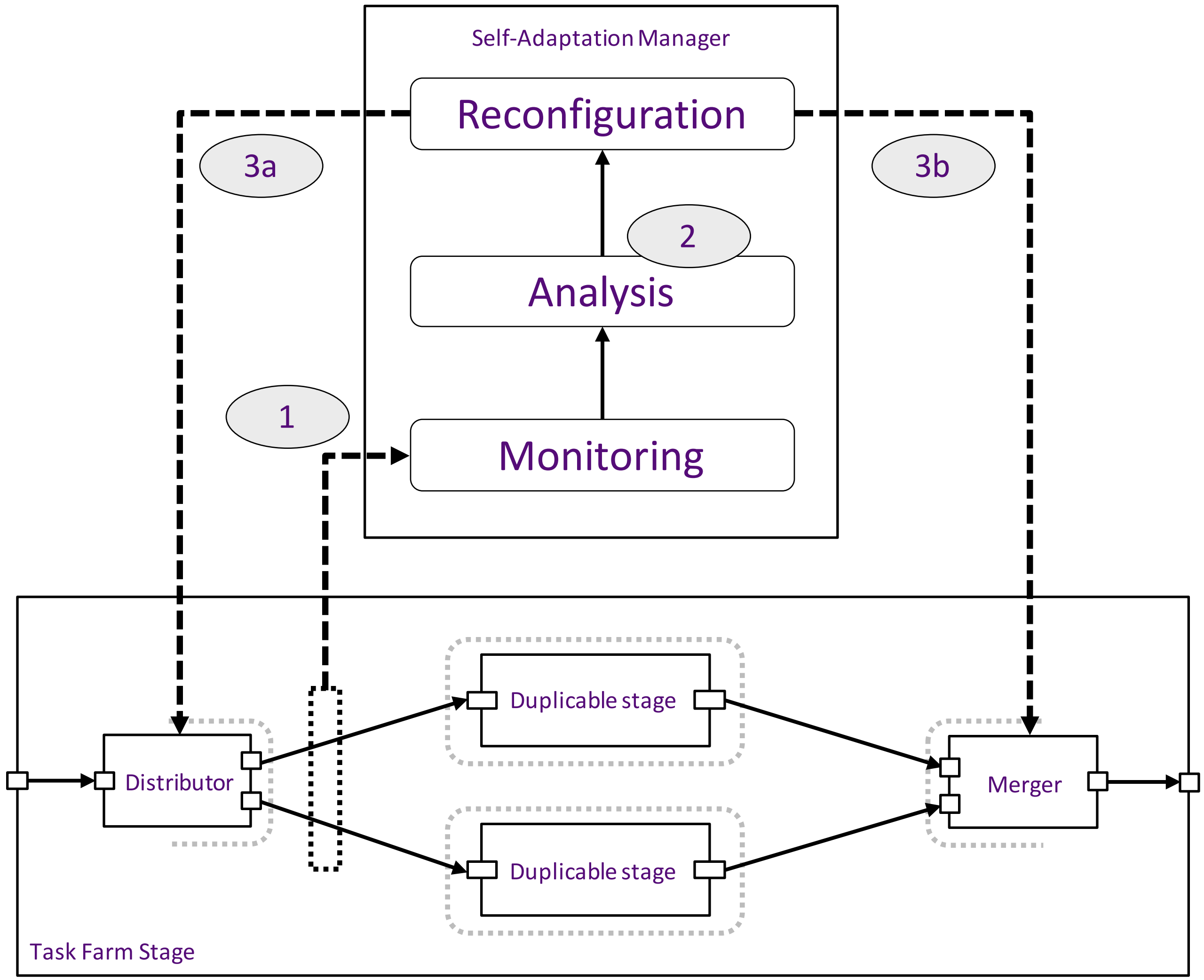
C|A|U Christian-Albrechts-Universität zu Kiel

Provides support for all task farm variations.

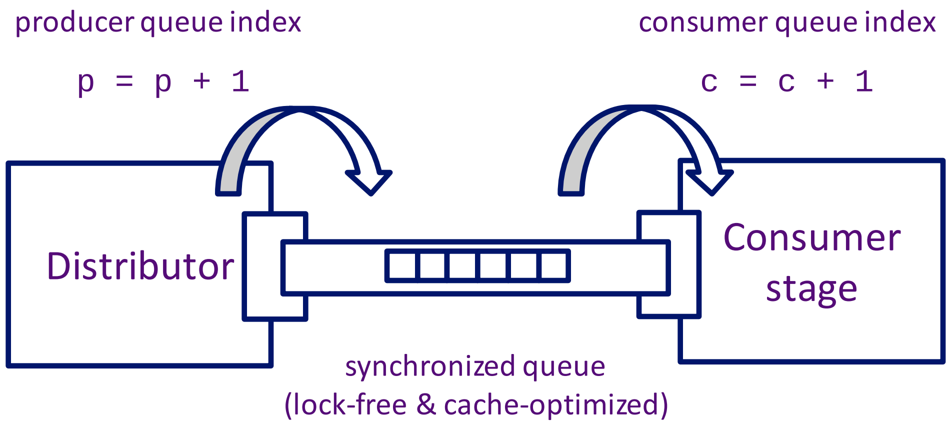


Our Self-Adaptation Manager (SAM)

Based upon MAPE-K [Kephart2003] and SLAstic [vanHoorn2014]



Monitoring Component



$$Consumer\ throughput\colon\ tp_c = c_t - c_{t-1}$$

where c_t is the consumer queue index at timestamp t

=> Monitoring has no performance influence on the threads executing the given P&F architecture

Analysis Component

Based upon [Ehlers2012, Rohr2015]

Throughput score

$$t_s = \frac{v-p}{v+p}$$

$-1.0 < t_s < 1.0$ and $v, p > 0$

v: most recent measurement

p: calculated predicted throughput based on recent history measurements

$t_s > 0 \Rightarrow$ more than expected

$t_s < 0 \Rightarrow$ less than expected

p is calculated by a throughput prediction algorithm:

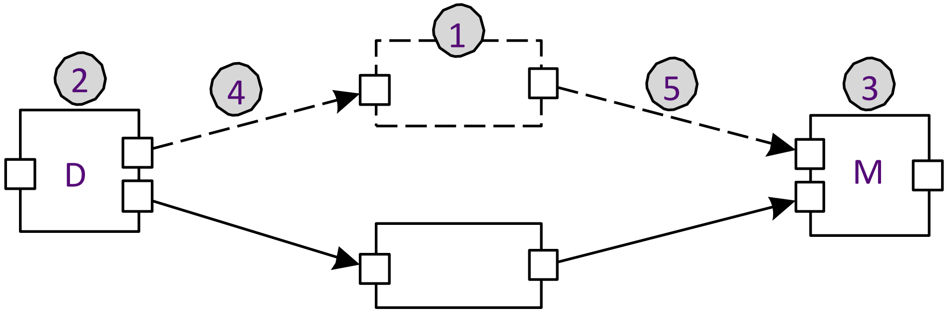
mean algorithm $\frac{1}{n}\sum_{i=1}^{n} p_i$

weighted algorithm $\frac{\sum_{i=1}^{n} \omega_i p_i}{\sum_{i=1}^{n} \omega_i}$ $\quad \omega_i > \omega_j$ *für* $i > j$

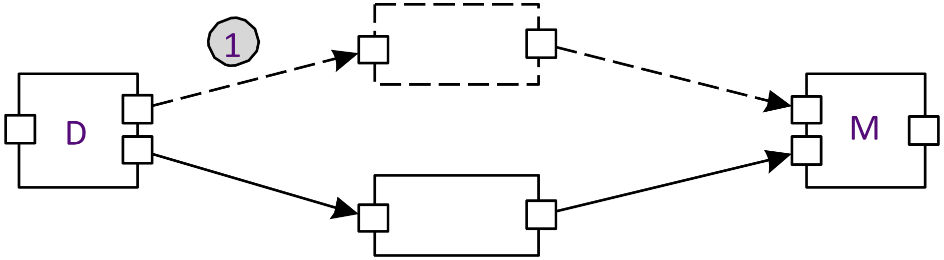
regression algorithm common least squares regression model

Reconfiguration Component

$t_s > tb_a$: add a stage



$t_s < -tb_r$: remove a stage



$t_s \in [-tb_r, tb_a]$: do nothing

tb_a : throughput boundary for addition, tb_r : throughput boundary for removal

Outline

Christian-Albrechts-Universität zu Kiel

- Motivation
- The Task Farm Parallelization Pattern
- Our Approach
- Evaluation
- Related Work
- Conclusions

Research Questions

- Feasibility

  1a) Does our TFS increase the overall throughput?

  1b) Does our SAM automatically adapt the number of stages according to the current runtime workload?

- Performance (Overhead)

  2a) To what extent does the throughput prediction algorithm influence the overall throughput?

  2b) To what extent does the throughput boundary influence the overall throughput?

Evaluation Setup

- 3 scenarios on 4 multi-core systems with 3 throughput prediction algorithms
- TFS implemented with our Java P&F framework

TeeTime

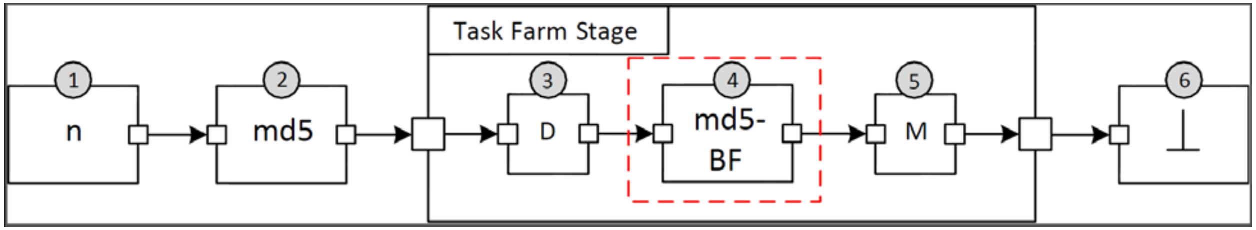
http://www.teetime-framework.net

- First-class entities: stage, pipe, port, configuration
- Support for pipelines, branches, feedback loops, stage composition
- Multi-threaded, high-throughput execution of stages

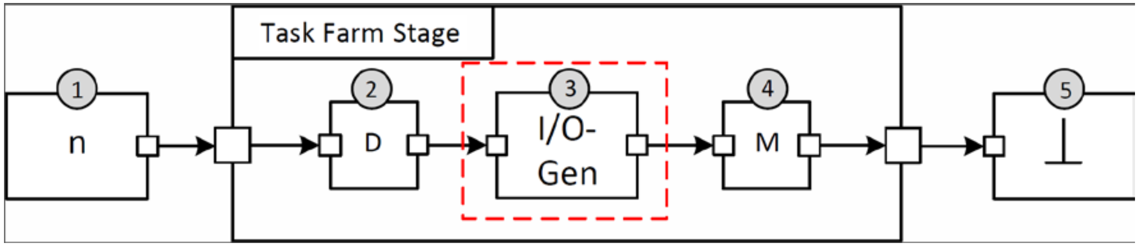
Evaluation Scenarios

balanced

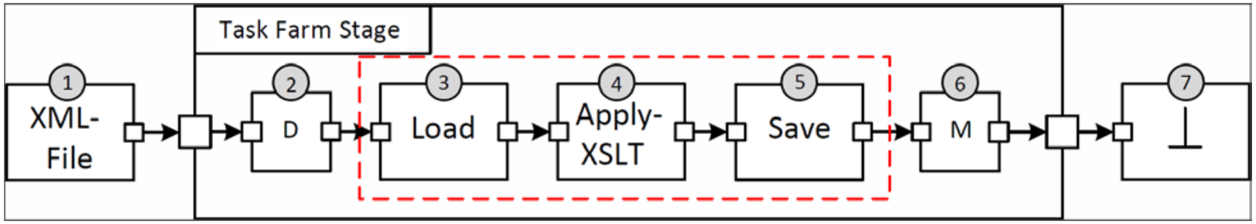
unbalanced



CPU-intensive scenario represented by Benchmark 1



I/O-intensive scenario represented by Benchmark 2



Combined CPU-I/O-intensive scenario represented by Benchmark 3

1a) Lowest Mean Execution Times

Christian-Albrechts-Universität zu Kiel

Benchmark configuration	Duration on *SUN* (w/o vs. w/ TFS)	Duration on *AMD-I* (w/o vs. w/ TFS)	Duration on *INTEL* (w/o vs. w/ TFS)	Duration on *AMD-II* (w/o vs. w/ TFS)
B1 (balanced workload)	21 sec./5 sec. = 4.2 boundary value = 0.025	10 sec./3 sec. = 3.3 boundary value = 0.025	17 sec./3 sec. = 5.7 boundary value = 0.025	25 sec./12 sec. = 2.1 boundary value = 0.2
B1 (unbalanced workload)	20 sec./5 sec. = 4.0 boundary value = 0.0	35 sec./7 sec. = 5.0 boundary value = 0.025	29 sec./4 sec. = 7.3 boundary value = 0.0	20 sec./10 sec. = 2.0 boundary value = 0.2
B2 (balanced workload)	13 sec./4 sec. = 3.3 boundary value = 0.025	49 sec./14 sec. = 3.5 boundary value = 0.225	15 sec./4 sec. = 3.8 boundary value = 0.025	26 sec./17 sec. = 1.5 boundary value = 0.2
B3 (balanced workload)	34 sec./7 sec. = 4.9 boundary value = 0.2	13 sec./4 sec. = 3.3 boundary value = 0.025	13 sec./2 sec. = 6.5 boundary value = 0.025	9 sec./5 sec. = 1.8 boundary value = 0.2

Table 1: Lowest mean execution times of the benchmark configurations achieved without and, respectively, with our TFS on the four multi-core systems. For each benchmark configuration, the regression prediction algorithm was used.

1b) Throughput w.r.t. Stages

Christian-Albrechts-Universität zu Kiel

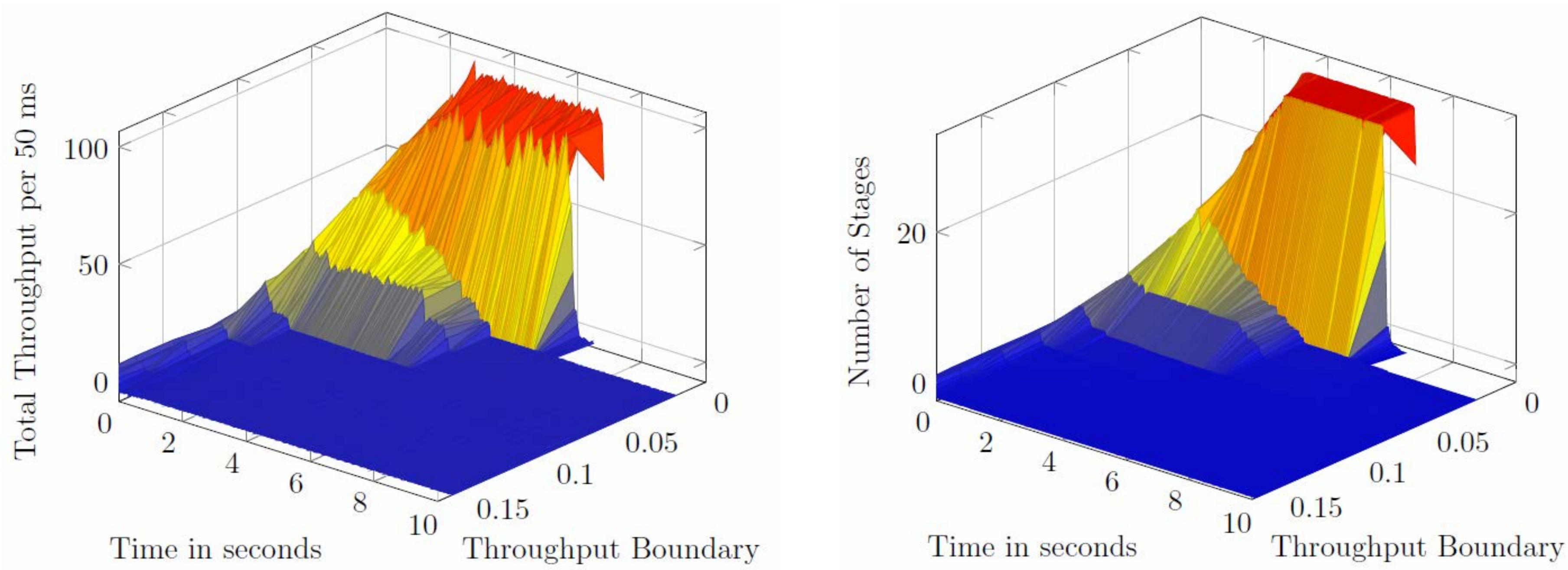
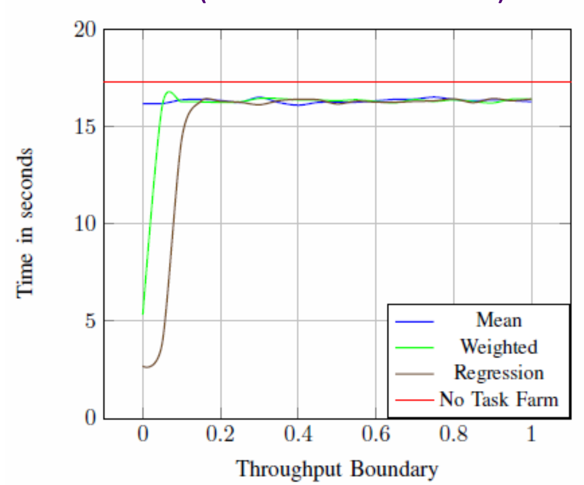


Figure 2: Benchmark 1 with a balanced workload on the Intel Xeon system

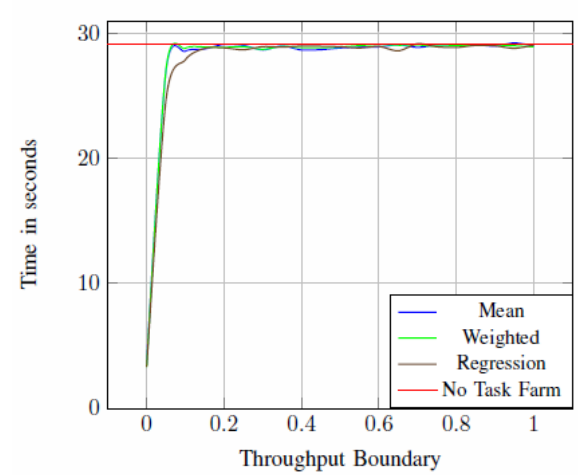
2a+2b) Performance Influences

Christian-Albrechts-Universität zu Kiel

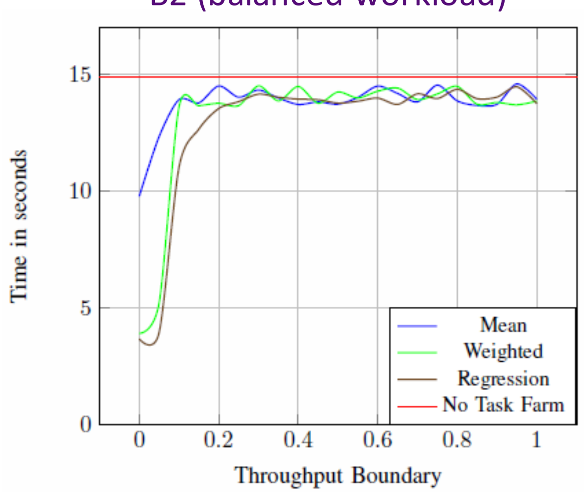
B1 (balanced workload)



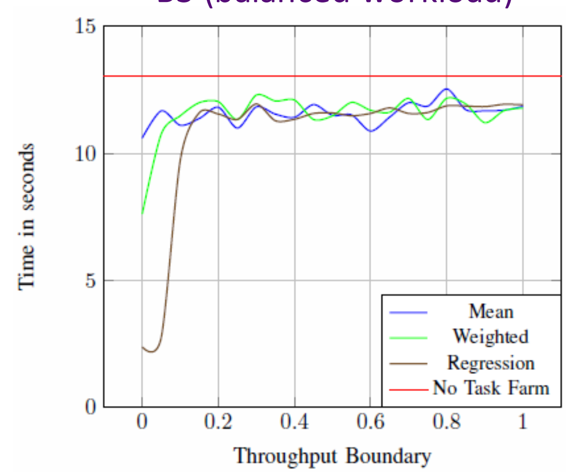
B1 (unbalanced workload)



B2 (balanced workload)



B3 (balanced workload)



Measurement results from the Intel Xeon system

Outline

Christian-Albrechts-Universität zu Kiel

- Motivation
- The Task Farm Parallelization Pattern
- Our Approach
- Evaluation
- Related Work
- Conclusions

Related Work

Related P&F-similar frameworks:

- FastFlow [Aldinucci2013]
- StreamIT [Thies2002]
- Pipes [http://www.tinkerpop.com]
- Akka [http://akka.io]

Related patterns:

- Map-Reduce [Dean2008]
- Fork-Join [Lea2000]

Self-adaptation in general:

- MAPE-K control loop [Kephart2003]
- Frameworks: Rainbow [Garlan+2004], AQuA [Diaconescu+2004], the Adaptive Server Framework [Gorton+2008], SLAstic [vanHoorn2009]

Self-adaptation in P&F achitectures:

- Training phase [Suleman2010]
- Thread stages and shader stages [Sugerman2009]

Outline

- Motivation
- The Task Farm Parallelization Pattern
- Our Approach
- Evaluation
- Related Work
- Conclusions

Conclusions

- Design & implemenation of a task farm stage and an associated self-adaptation manager
- Evaluation of the feasibility and the performance (speedups up to 7.3)
- Best: regression algorithm with a „low“ boundary
- Replication package [doi: 10.5281/zenodo.46776] with all data and code provided

TeeTime

http://www.teetime-framework.net

Future work:

- Speedup sensitive to throughput boundary => Automatic identification at runtime
- Extend the duplicable interface to more than one input/output port
- More throughput prediction algorithms, e.g., ARIMA and Random
- Comparison of related approaches

References

[Parnas1972] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," Commun. ACM, vol. 15, no. 12, 1972.

[Rayside2006] D. Rayside, L. Mendel, and D. Jackson, "A dynamic analysis for revealing object ownership and sharing," in Proceedings of the International Workshop on Dynamic Systems Analysis. ACM, 2006.

[Cole1991] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, 1991.

[Aldinucci+1999] M. Aldinucci and M. Danelutto, "Stream Parallel Skeleton Optimization," in Proceedings of the International Conference on PDCS, 1999.

[Kephart2003] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, pp. 41–50, Jan 2003.

[vanHoorn2014] A. van Hoorn, Model-Driven Online Capacity Management for Component-Based Software Systems, ser. Kiel Computer Science Series. Kiel, 2014, no. 6, dissertation, Faculty of Engineering, Kiel University.

[Ehlers2012] Ehlers, Jens, *Self-Adaptive Performance Monitoring for Component-Based Software Systems, dissertation*, Kiel University, 2012, 252 pp

[Rohr2015] Rohr, Matthias, *Workload-sensitive Timing Behavior Analysis for Fault Localization in Software Systems*, dissertation, Faculty of Engineering, Kiel University, 2015, 224 pp

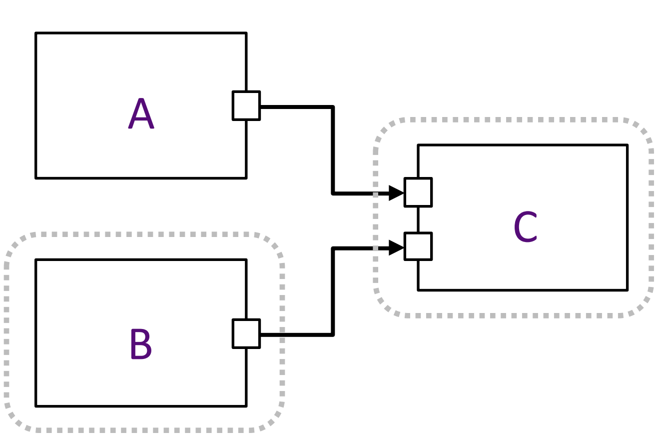
[Garlan+2004] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," Computer, vol. 37, no. 10, pp. 46–54, Oct 2004.

[Diaconescu+2004] A. Diaconescu, A. Mos, and J. Murphy, "Automatic performance management in component based software systems," in Proceedings of the International Conference on Autonomic Computing, May 2004.

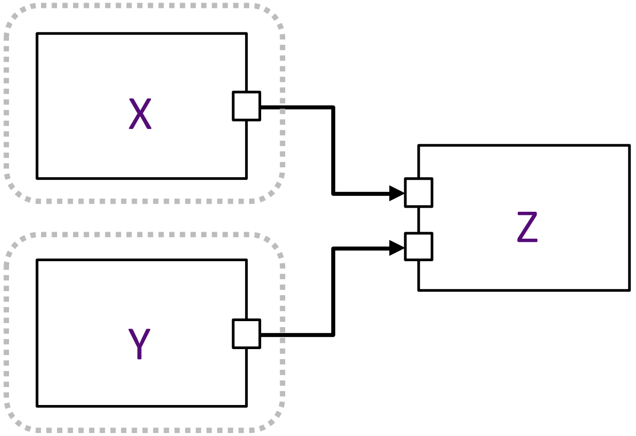
[Gorton+2008] I. Gorton, Y. Liu, and N. Trivedi, "An extensible and lightweight architecture for adaptive server applications," Software: Practice and Experience, vol. 38, no. 8, 2008.

Unknown Stage Responsibility

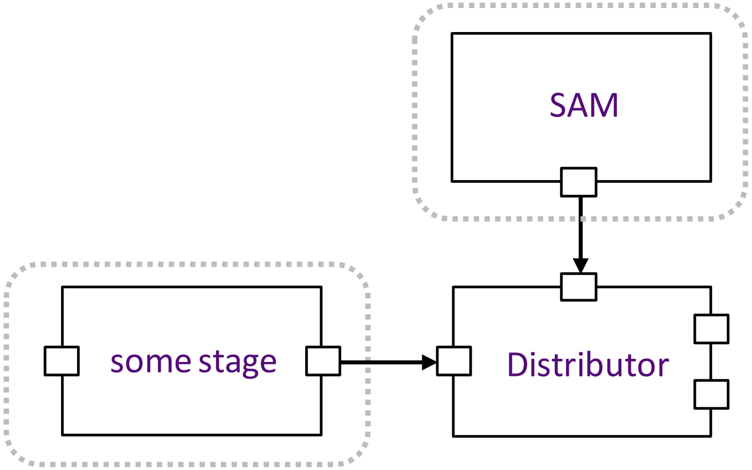
Christian-Albrechts-Universität zu Kiel



Conflict: it is unclear whether the thread of B or C should execute the passive stage A.



Conflict: it is unclear whether the thread of X or Y should execute the passive stage Z.



Analysis Component

Throughput score

$$t_s = \frac{v-p}{v+p} \qquad -1.0 < t_s < 1.0, \quad v > 0$$

v: most recent measurement (always positive)

p: calculated predicted throughput based on recent history measurements

$\frac{30-10}{30+10} = \frac{1}{2} > 0 \Rightarrow$ more than expected $\qquad \frac{30-50}{30+50} = -\frac{1}{4} < 0 \Rightarrow$ less than expected

p is calculated by a throughput prediction algorithm:

mean algorithm $\quad \frac{1}{n}\sum_{i=1}^{n} p_i$

weighted algorithm $\quad \frac{\sum_{i=1}^{n} \omega_i p_i}{\sum_{i=1}^{n} \omega_i} \qquad \omega_i > \omega_j \text{ für } i > j$

regression algorithm $\quad$ (least squares regression model)

Used Multi-Core Systems

Christian-Albrechts-Universität zu Kiel

System	*SUN*	*AMD-I*	*INTEL*	*AMD-II*
# Processors	2	2	2	1
Processor	UltraSPARC T2+	AMD Opteron 2384	Intel Xeon E5-2650	AMD Opteron 2356
Architecture	SPARC V9 (64 Bit)	x86-64	x86-64	x86-64
Clock/Core	1,4 GHz	2,7 GHz	2,8 GHz	2,3 GHz
Cores per processor (hardware threads)	8 (64)	4 (4)	8 (16)	4 (4)
RAM	64 GB	16 GB	128 GB	4 GB
Disk Controller	RAID1/SAS	RAID1/SATA	SATA	RAID1/SATA
OS	Solaris 10	Debian 8	Debian 8	Debian 7