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

Christian Wulf
Software Engineering Group
Kiel University
24098 Kiel, Germany
Email: chw@informatik.uni-kiel.de

Christian Claus Wiechmann
Software Engineering Group
Kiel University
24098 Kiel, Germany
Email: ccw@informatik.uni-kiel.de

Wilhelm Hasselbring
Software Engineering Group
Kiel University
24098 Kiel, Germany
Email: wha@informatik.uni-kiel.de

***Abstract*—The Pipe-and-Filter style represents a well-known family of component-based architectures. By executing each filter on a dedicated processing unit, it is also possible to leverage contemporary distributed systems and multi-core systems for a high throughput.**

However, this simple parallelization approach is not very effective when (1) the workload is uneven distributed over all filters and when (2) the number of available processing units exceeds the number of filters. In the first case, parallelizing all filters can lead to a waste of resources since only the slowest filter is responsible for the overall throughput. In the second case, some processing units remain unused.

In this paper, we present an automatic parallelization approach providing high throughput and utilizing the available processing units. Our main idea is to provide a composite filter that is wrapped around an existing filter to increase its throughput. We call this composite filter the *Task Farm Filter* since it implements the Task Farm parallelization pattern. It creates and executes multiple instances of the underlying filter in parallel. Moreover, we present a modular, self-adaptive mechanism that automatically adapts the number of instances at runtime to achieve the highest possible throughput.

Finally, we present an extensive experimental evaluation of our self-adaptive task farm filter by employing a CPU-intensive, an I/O-intensive, and a hybrid scenario. The evaluation shows that our task farm automatically parallelize the underlying filter and thus increases the overall throughput. Furthermore, the evaluation shows that our task farm scales well with the workload of the executed Pipe-and-Filter architecture.

***Index Terms*—Parallel processing, Software performance, Software architecture**

I. Introduction

The Pipe-and-Filter (P&F) architectural style [1]–[3] divides a complex task into several successive subtasks such that each of them is implemented by a separate, independent so-called *filter*. Filters communicate with each other by transferring data via *pipes*. An example P&F architecture is shown in Figure 1. It represents Parnas' *Keyword In Context* program [4] as P&F implementation [5]. It reads a text file, produces circular shifts of the lines, alphabetizes these shifts, and prints out the results.

With the use and adoption of big data, P&F gained an increased popularity both in industry and in research. Recent research [6]–[9] addresses the problem of how to leverage and to optimize contemporary multi-core systems for a high throughput. One simple approach is to execute each filter concurrently. However, we always need to ensure that the computation effort outweighs the communication effort. Otherwise the performance of a multi-core implementation could even become worse than the performance of a single-core implementation due to additional synchronization costs. Moreover, an unevenly distributed workload can lead to a waste of resources as only the slowest filter determines the overall throughput. Finally, if there are less filters than processing units available, the underlying hardware is not fully utilized.

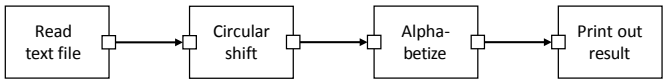


Fig. 1: An example pipeline: Parnas' *Keyword In Context* program [4] as P&F implementation [5]

In this paper, we propose an automatic parallelization approach to increase the throughput of P&F architectures on contemporary multi-core systems. Our main idea is to provide a composite filter that is wrapped around an existing filter. It automatically duplicates the underlying filter and executes their instances in parallel. We call this composite filter the *Task Farm Filter* since it implements the Task Farm parallelization pattern [6]. In this way, it is possible to parallelize as many instances of the slowest filter as processing units are available.

In addition to the task farm filter, we also propose an associated self-adaptation manager. This manager is able to automatically adapt the number of filter instances at runtime based on the current throughput of the given task farm filter. In this way, a P&F architecture can achieve a high throughput even under an unevenly distributed workload. For example, if the workload is higher or lower than expected, the manager automatically increases or, respectively, decreases the number of filter instances. In particular, the manager is able to remove a filter instance if such instance does not contribute to the overall throughput anymore. Thus, the manager simultaneously tries to use as few processing units as possible.

We implemented both the task farm filter and the self-adaptation manager with our Java-based P&F framework TeeTime [10]. Everything is available as open-source. Furthermore, we provide all data and results of our experiments as replication package [11] for review and evaluation purposes.

Structure of this paper: First, we present some foundations of the P&F architectural style (Section II) and of the task farm parallelization pattern (Section III). Then, we present our task farm filter in Section IV. Afterwards, we describe our self-adaption manager in Section V. In Section VI, we present an extensive experimental evaluation of our task farm filter including the self-adaption manager. Finally, we discuss related work in Section VII and conclude this paper in Section VIII.

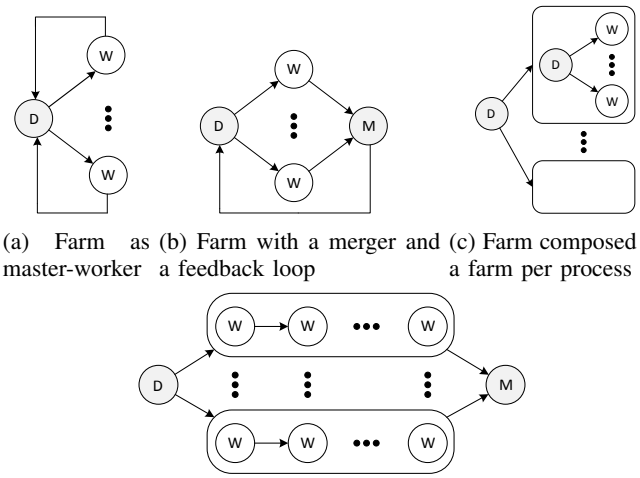
II. The Pipe-and-Filter Architectural Style

Mary Shaw is one of the first authors who explicitly described P&F architectural style. In her work [1], she describes it as a useful system organization that consists of filters accepting one stream of inputs and emitting one stream of outputs. In general, a filter is defined by its name, its execution logic, and its input and output ports [12]. Each port is typed such that it accepts or sends data of a particular type only. One execution of a filter proceeds as follows: It reads data from one or more of its input ports, transforms that data, and writes the results to one or more of its output ports. Hence, the P&F style also allows to model feedback loops [13] and branches [14]. For modularity and re-usability reasons, a filter can be composed of predefined child filters.

III. The Task Farm Parallelization Pattern

In 1991, Cole [15] introduced so-called *Algorithmic Skeletons*. Such a skeleton is defined as a higher-order function describing a certain computational behavior. One example is the Task Farm parallelization pattern which has been defined by, e.g., Aldinucci and Danelutto [16]. Semantically, it describes the identity function of its underlying algorithm. However, the input data stream is parallelized such that the underlying algorithms is concurrently executed. In [6], the task farm pattern was, among other patterns, implemented for *FastFlow*, a framework for streaming applications where stages have exactly one input port and one output port [17]. In the following, we briefly describe the task farm parallelization pattern. For a more detailed description, we refer to [16] and [18].

Context & Problem We intend to perform an operation f on each task of a given stream of tasks as fast as possible. For this purpose, we intend to execute f on multiple tasks in a concurrent way. First, a task distributor is required to distribute tasks to worker processes. Depending on the scenario, different distribution strategies are possible, e.g., round-robin or broadcasting. Second, it is important to determine how many worker processes should run concurrently. Hence, we need to find a balanced ratio between the computation cost of f and the communication cost between the distributor and the workers. Finally, we need to consider what we should do with the results of the worker processes. For example, the results could be fed back to the distributor or merged for later processing.



(a) Farm as master-worker (b) Farm with a merger and a feedback loop (c) Farm composed of a farm per process

(d) Farm composed of a pipeline per process

Fig. 2: Different manifestations of the task farm pattern. (D) means distributor, (W) means worker, and (M) means merger.

Solution In order to support arbitrary scenarios, the task farm parallelization pattern provides a generic task distributor which can be declared active or passive. In the former case, it autonomously distributes incoming tasks to one or more worker processes according to a user-defined distribution strategy. In the latter case, it serves as a task pool from which worker processes fetches tasks.

Figure 2 shows some well-known manifestations of the task farm pattern. For example, Figure 2a illustrates a version which is also called the master/worker pattern. Here, the distributor is responsible for both the distribution of the tasks to the workers and the collection of the result from the workers. Figure 2b illustrates a more modular version where the collection of results is extracted from the distributor to a so-called merger. Optionally, the merger can still feedback its result to the distributor. Figure 2c shows an example for a composite worker. Here, the workers are again composed of a task farm. Similarly, Figure 2d shows a another composite worker which are composed of a worker pipeline.

Typically, the number of worker processes is fixed and given in an initialization phase (e.g., see FastFlow [17]). However, the task farm pattern does not forbid to adapt the number of worker processes at runtime. Cloud environments, e.g., use the master-worker version to scale their instances according to the workload. In Section V, we describe how we combine a self-adaptation manager and the task farm pattern to allow a self-adaptive behavior of P&F applications.

IV. Overview of the Task Farm Stage

In this section, we present our approach to increase the throughput of P&F architectures. Although it can also be applied to distributed systems, we have only implemented and evaluated it for the execution on multi-core systems so far. Our main idea is to provide a composite filter that is wrapped around an existing filter in order to parallelize it. We call this

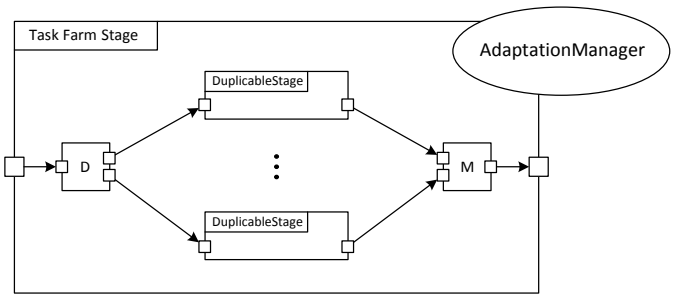


Fig. 3: Architecture of our task farm stage

composite filter the Task Farm Stage (TFS) since it utilizes the Task Farm Parallelization Pattern [16]. The structure of the TFS is illustrated in Figure 3. For the rest of this paper, we will use the term *stage* as generalization for *data sources*, *filters*, and *data sinks*, as categorized by Buschmann et al. [19].

The TFS is a composite stage with additional parallelization functionality. Its child stages are the Dynamic Distributor (shown as *D* in Figure 3), the Dynamic Merger (shown as *M* in Figure 3), and the Duplicable Stage. The Duplicable Stage represents the stage which should be parallelized. Usually, there are multiple instances of the Duplicable Stage at runtime each running in a dedicated thread. Additionally, a self-adaptation manager is used to monitor the TFS and to maximize its performance (see Section V).

In general, the task farm parallelization pattern can be applied to all P&F architectures. However, our TFS currently defines the following assumptions regarding the P&F architecture to limit the complexity:

- The TFS in general and the Duplicable Stage in particular may not contain feedback loops. For example, output ports of a Duplicable Stage must not lead to the distributor of its own TFS. This limitation allows us to measure the pipe throughputs more accurately, which is necessary for the self-adaptation manager (see Section V).
- Each Duplicable Stage has exactly one input port and one output port. More input and output ports would possibly require more input and output ports for the TFS itself, leading to a higher complexity.

Since the Duplicable Stage may be a composite stage, our TFS covers the task farm pattern versions 2b-2d from Figure 2.

A typical traversal of any data element arriving at the input port of the TFS is as follows. At first it arrives at the Dynamic Distributor. The distributor chooses which instance of the Duplicable Stage (worker stage) is going to process the element according to a specified distribution strategy. Afterwards, the distributor sends the element to the output port leading to the chosen worker stage. The worker stage then processes the data element and finally sends it to the Dynamic Merger. The task of the merger is to merge incoming elements to a single output stream. The concrete behavior of the merger and the order of the elements in the output stream depends on the specified merging strategy.

In the following sections, we discuss the components of the TFS. In Section IV-A, we explain the behavior for the Dynamic Distributor and the Dynamic Merger as well as some of their strategies. Section IV-B addresses the definition of the Duplicable Stage. Section IV-C explains the process of adding and removing a worker stage at runtime.

A. Dynamic Distributor & Dynamic Merger

The Dynamic Distributor and the Dynamic Merger have to process all data elements entering the TFS. Therefore, an efficient implementation of these stages is a key requirement to an efficient TFS. Furthermore, as we intend the self-adaptation manager to be able to dynamically add or remove worker stages, the distributor and merger have to be able to add and remove ports at runtime.

As mentioned above, the task of the Dynamic Distributor in the TFS is the distribution of each data element to a worker stage. We can define an arbitrary strategy for the distributor to define its exact behavior. In the following, we discuss three possible distribution strategies.

1) CloneStrategy: The distributor duplicates each incoming data element according to the current number of worker stages, before sending an exact copy to each worker stage. Therefore, all worker stages produce the same output elements which are then merged and passed to the output port of the TFS. Although this strategy trivially increases the overall throughput, it also violates the original semantics of the Duplicable Stage.

2) BlockingRoundRobinStrategy: This strategy causes the distributor to send the current input element to the next worker stage selected in round-robin order. Hence, each worker stage has approximately the same workload, assuming the processing time is the same for each data element. However, as the capacity of pipes is often bounded, a distribution strategy must also handle the case of a full pipe. This strategy waits until the pipe is free again, which can waste time if other worker stages have non-full input pipes.

3) NonBlockingRoundRobinStrategy: This strategy behaves almost exactly like the BlockingRoundRobinStrategy. However, if the input pipe of the chosen worker stage is full, it searches for another worker stage in round-robin order. Therefore, this distribution strategy provides the best performance.

Although the task of the Dynamic Merger is the opposite of the task of the Dynamic Distributor, it uses very similar concepts regarding its merging strategy. In the following, we discuss two possible merging strategies.

1) BlockingRoundRobinStrategy: This strategy behaves almost analogous to its counterpart of the Dynamic Distributor. The only difference is that this strategy skips input ports which are no longer in use (e.g., if the corresponding worker stage has been removed). Otherwise, we would cause a livelock or, respectively, a deadlock depending on whether this strategy is implemented in a busy-waiting or in a blocking-read way.

2) NonBlockingRoundRobinStrategy: This strategy behaves analogous to its counterpart of the Dynamic Distributor. It searches for the next non-empty input port in round-robin order and passes the corresponding element to the merger's

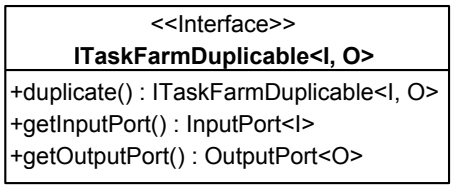


Fig. 4: If a stage implements `ITaskFarmDuplicable`, our TFS is able to duplicate it and to execute it in parallel.

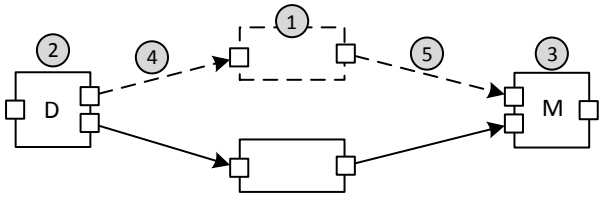


Fig. 5: Duplicating a stage: (1) new worker stage, (2) distributor, (3) merger, (4) new worker stage input pipe, (5) new worker stage output pipe

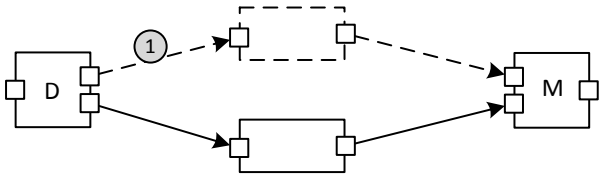


Fig. 6: Removing a duplicated stage: (1) worker stage input pipe to be removed

output port. Due to the absence of any blocking mechanism, this merging strategy provides the best performance.

Additionally, the Dynamic Distributor and the Dynamic Merger need a way to dynamically add and remove output and input ports, respectively. Therefore, we introduce two port actions which enable the distributor and merger to provide this functionality. The first action triggers the distributor or the merger to add a new port to itself and to connect it with the new worker stage. The second action triggers the distributor or the merger to remove an existing port from itself.

Both the distributor and the merger provide a port action interface to the self-adaptation manager. In this way, the self-adaptation manager can dynamically add or remove worker stages from the TFS depending on the workload.

B. Duplicable Stage

The TFS can parallelize an arbitrary (composite) stage if the stage is a Duplicable Stage, i.e., if it implements the interface `ITaskFarmDuplicable` shown in Figure 4. The interface requires to implement three methods.

The two methods `getInputPort` and `getOutputPort` are necessary to retrieve the input port and the output port of the Duplicable Stage. The TFS requires access to these ports to properly connect the worker stages to the Dynamic Distributor and the Dynamic Merger.

The `duplicate` method is a crucial part of the interface. It generates an additional worker stage from the corresponding instance of the Duplicable Stage. Thus, the user of the TFS can freely implement the duplication behavior of the Duplicable Stage, providing a way to implement most use cases. The duplication method is called whenever the self-adaptation manager decides to add another worker stage to the TFS.

The programming effort to migrate an existing stage to a duplicable stage is low. It only needs to implement the interface `ITaskFarmDuplicable`. First, the methods `getInputPort()` and `getOutputPort()` must return one input port and, respectively, one output port of the existing stage. Second, the method `duplicate()` must return a new copy of the existing stage. If the stage is stateless, an invocation of the constructor is sufficient. Otherwise, the implementation must also ensure that the internal and shared state attributes are copied in a correct and thread-safe way.

C. Addition and Removal of Worker Stages

The TFS achieves its parallelization by dynamically adding and removing worker stages. The self-adaptation manager monitors the workload and decides whether further parallelization is reasonable. For this purpose, the TFS requires to implement some logics for the dynamic addition and for the dynamic removal of the worker stages.

The process to duplicate a worker stage is illustrated in Figure 5. The first step is to create a new worker stage (1) by using the duplication method on an arbitrary existing worker stage inside the TFS. To prevent race conditions between the new worker stage and its not yet exiting input and output pipes, it does not yet process data elements. To connect the Dynamic Distributor (2) and the Dynamic Merger (3), we then create the worker stage input pipe (4) between the distributor and the worker stage. As discussed in Section IV-A, we use the CreatePortAction to connect the pipe with a new output port of the distributor. Similarly, the worker stage output pipe (5) between the worker stage and the merger is created. The pipe is then also connected to the merger via the CreatePortAction. Finally, the new worker stage (1) is started in a dedicated thread and begins processing elements. In this way, it increases the total throughput of the TFS.

The process to remove an existing worker stage is illustrated in Figure 6. The only step needed to remove an existing worker stage is to deactivate the distributor's output port which is connected to the worker stage's input pipe (1). Afterwards, the distributor will no longer send data elements to the removed worker stage, allowing it to safely process all buffered elements. If the removed worker stage has no further elements to process, it terminates itself and closes its output port. Since the merger uses the BusyWaitingRoundRobinStrategy (see Section IV-A), it detects the closed port and proceeds with another worker stage.

It is vital that the removal of a worker stage allows the removed stage to work off all remaining elements buffered by the input pipe. Otherwise, the TFS might loose some elements whenever a worker stage is removed.

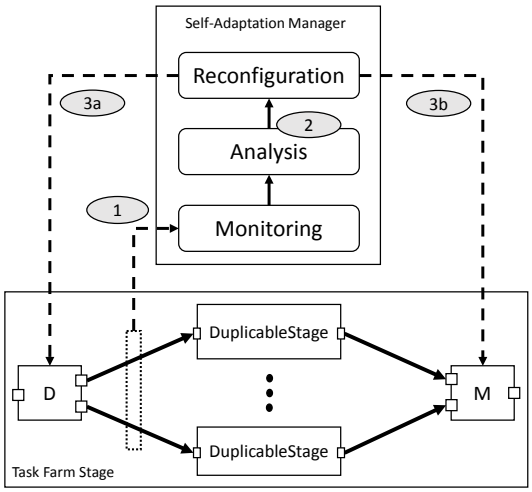


Fig. 7: The SAM and how it interacts with the TFS: the Monitoring component measures the throughput of the pipes (1), the Analysis component computes a corresponding throughput score (2), and the Reconfiguration component sends actions to the distributor (3a) and to the merger (3b) to adapt the TFS.

Because a TFS has multiple worker stages at runtime, it has to decide which available stage should be removed. We choose the strategy to always remove the worker stage whose input pipe has the least number of buffered data elements. Since the lower the number of buffered elements, the less is the time to wait for the stage to be actually removed. Hence, this strategy usually provides a low latency for removing worker stages.

V. Structure of the Self-Adaptation Manager

The task of the Self-Adaptation Manager (SAM) is to control the addition and removal of worker stages in its corresponding TFS. Therefore, each TFS inside the P&F architecture requires a dedicated SAM. The design of the SAM is based on the general design of an adaptable software system described by van Hoorn et al. [20] and is shown in Figure 7.

The SAM consists of three components. The Monitoring component monitors the throughput of the pipes which connect the Dynamic Distributor and the worker stages with each other. In this way, we measure the performance of each worker stage. Afterwards, the Analysis component analyzes the measurements of the Monitoring component. It calculates how much the throughput of the TFS has changed since the last few measurements. The last component is the Reconfiguration component, which takes the result of the Analysis component and decides whether the TFS should add or remove a worker stage. The cycle as shown in Figure 7 is then completed and starts again with the Monitoring component after a user-defined delay (our default is 50 ms). In Section V-A to V-C, we introduce each of the three components in more detail.

A. Monitoring Component

As mentioned above, the Monitoring component measures the throughput of the pipes between the Dynamic Distributor and the worker stages (see (1) in Figure 7). We compute the throughput of a single pipe by n/td where n is the amount of elements pulled from the pipe since the last SAM cycle and td is the time difference between the current and the last SAM cycle. This throughput definition directly reflects the actual productivity of the corresponding worker stage without being influenced by element buffering or similar issues.

The Monitoring component saves the sum as well as the average of the throughputs of the worker stages. We save these values for later use to make informed decisions on whether we can further optimize the TFS performance by adding or removing a worker stage. As we collect these measurements in every SAM cycle, we construct a history of measurements.

B. Analysis Component

The Analysis component analyzes the most recent measurement and the measurements of the Monitoring component to calculate a so-called *throughput score*. The throughput score serves as action indicator so that the Reconfiguration component can decide whether it should add a worker stage, remove a worker stage, or do nothing.

Let $v \in \mathbb{N}$ be the most recent measurement. Let $p \in \mathbb{N}$ be a calculated predicted throughput based on a number of recent history measurements. The throughput score $ts \in \mathbb{R}$ with $-1.0 < ts < 1.0$ is then defined by

$$ts = \frac{v - p}{v + p}.$$

For example, let $v = 3$ and $p = 1$, i.e., the throughput has increased since the last measurement. The corresponding throughput score is $ts = \frac{v-p}{v+p} = \frac{3-1}{3+1} = 0.5$. In general, a positive ts describes a throughput increase, a negative ts describes a throughput decrease. Moreover, the higher $|ts|$ is, the more definitive is the change in the throughput.

To calculate p, we can choose one of multiple prediction algorithms, each providing some unique characteristics regarding their forecasting behavior.

1) Mean Algorithm: The mean algorithm uses a number of previous throughput measurements of the TFS and calculates the average value of it. This value will then be interpreted as the expected value of the current point in time.

This is one of the least complex algorithms that can be used to predict values. While it is very fast due to its low computational effort, it does not produce acceptable results for our use case. If the throughput is constant, the average of the last few measurements is always a correct prediction, leading to correct forecasts. However, the algorithm looses this advantage for every other runtime behavior and cannot accurately predict future throughput measurements.

2) Weighted Algorithm: The weighted algorithm is a variation of the mean algorithm. It also computes the average of a certain number of measurements, but it additionally adds weights in such a way that more recent measurements have more impact on the prediction.

For our use case, this algorithm is more usable than the mean algorithm since it reacts faster on changes of the

throughput. However, the algorithm does not behave well for linearly and exponentially growing throughputs since it does not extrapolate any behavior of the throughput measurements. For an irregular runtime behavior, it in turn yields comparatively good results since extrapolation is not possible at all.

3) Regression Algorithm: The regression algorithm uses a statistical regression algorithm to predict the throughput at the current point in time. It uses at least two previous measurements to construct a straight line $y = ax + b$, where x is a point in time, y the throughput at that time, and $a, b \in \mathbb{R}$. If more than two measurements are used, a straight line is found that corresponds best to all provided data points. The prediction can be obtained by solving the equation by setting x to the current point in time.

This algorithm behaves very well for any nearly linear runtime behavior of the TFS. Exponential and other regular behavior can also be accurately predicted by using a lower amount of data points for the line construction. This is possible since exponential functions are mathematically nearly linear for a small interval. However, since the regression algorithm assumes a linear function in the runtime behavior, it can yield very imprecise predictions for irregular behavior.

C. Reconfiguration Component

The Reconfiguration components directly controls the TFS. It decides, depending on the throughput score calculated in the Analysis component, whether a worker stage should be added, removed, or kept running. For this purpose, the component communicates with the distributor and with the merger by sending actions to them (see (3a) and (3b) in Figure 7). There are the following three different actions.

The *add action* represents the addition of a new worker stage, resulting in a higher degree of parallelization of the Duplicable Stage. This action is chosen if the throughput score was positive and above a given throughput boundary, i.e., the addition of the last worker stage gained a performance boost.

The *remove action* represents the removal of a worker stage. It is triggered in two situations: either when the throughput score has become so low that further parallelization is not likely to provide an increase in performance, or when the workload on the TFS has become so low that not all worker stages are utilized anymore. By removing such a worker stage, the associated processing unit is released. In this way, we avoid an inefficient usage of the processing units. Moreover, we reduce the communication overhead of the TFS' internal components.

The third and last action is the *no-op action*. It is useful whenever we do not have enough data to reliably decide another action. This situation occurs if the time between SAM cycles is very low. In that case, the worker stages might not have had enough time to produce new output elements.

VI. Evaluation

In this section, we evaluate our task farm stage (TFS) and the associated self-adaptation manager (SAM). Our goals and corresponding questions are as follows:

Feasibility: we intend to increase the overall throughput of a P&F architecture by applying our approach.

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: we intend to maximize the overall throughput of a P&F architecture when using our TFS and our SAM.

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?

A. Scenarios

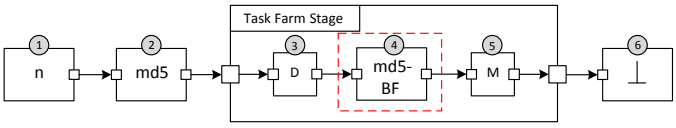
We consider four different scenarios in order to evaluate our goals. The first scenario represents a CPU-intensive computation with a balanced workload. Similarly, the second scenario represents a CPU-intensive computation, though with an unbalanced workload. The third scenario represents an I/O-intensive computation where, e.g., the file system is accessed most of the time. The fourth scenario represents a more common and more realistic kind of computation. It covers both a CPU-intensive part and an I/O-intensive part. We implemented each scenario as a benchmark. Figure 8a-8c show their corresponding P&F architectures. While Benchmark 1 and 2 use a single stage as duplicable stage, Benchmark 3 uses a composite duplicable stage with 3 inner stages (cf. Figure 2d).

For the implementation, we used the Java-based P&F framework TeeTime [10]. TeeTime allows to model and to execute arbitrary P&F architectures. For example, it supports feedback loops, multiple input/output ports per stage, and the composition of several stages to a single one. In particular, TeeTime is able to utilize contemporary multi-core systems by executing stages of a P&F architecture in parallel. Hence, we chose this framework for our evaluation.

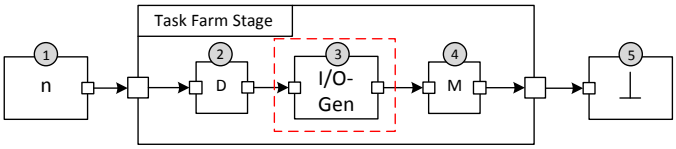
Benchmark 1 (B1) uses the task farm to compute the original number for a given hash value and a fixed hash function by applying bruteforcing (see Stage 4 in Figure 8a). It generates the hash value for every number (from 0 to $max(integer)$) until it matches the input hash value. Then, it outputs the corresponding number. Stage 1 serves as a workload generator for a sequence of either a fixed number (balanced workload) or a linearly increasing number (unbalanced workload). Stage 2 computes the hash value of each incoming number and serves as input for the task farm. Stage 6 represents a sink which discards the incoming original numbers.

Benchmark 2 (B2) uses the task farm to write n characters to a temporary text file for a given number n (see Stage 3 in Figure 8b). Similar to B1, Stage 1 and 6 serve as a balanced work load generator and as a sink, respectively.

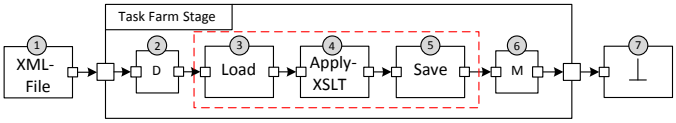
Benchmark 3 (B3) uses the task farm to transform XML files by applying a fixed XSLT transformation (see Figure 8c). Stage 3 loads the incoming XML file, Stage 4 transforms it in memory, and Stage 5 writes it back to the file system. Again, Stage 1 and 7 serve as a balanced work load generator and as a sink, respectively. For benchmarks details, we refer to our replication package [11] and to our TeeTime-Project [21].



(a) Benchmark 1 represents a CPU-intensive computation with either a balanced workload (using a constant workload generator) or an unbalanced workload (using a linearly increasing workload generator)



(b) Benchmark 2 represents an I/O-intensive computation with a balanced workload



(c) Benchmark 3 represents a combined CPU-I/O-intensive computation with a balanced workload

Fig. 8: Benchmarks for the performance evaluation. Each red rectangle represents the duplicable stage of the task farm stage.

B. Experimental Setup

We executed each benchmark on four different multi-core systems. Table I shows their hardware and software details. Each benchmark was implemented with and executed by TeeTime 2.0. We set the iteration interval of our SAM to 50 ms in order to measure a throughput that is greater than one for all scenarios on all systems. We configured the workload generator stages to produce 1,000 to 10,000 elements depending on the benchmark and the system. Since we do not intend to compare benchmark configurations or systems with each other, we chose different values for each pair. In this way, we effectively limited the execution times of the benchmarks.

We use the Oracle Java Runtime Environment (JRE)[1] in the version *1.8.0_60-b27*. We set the number of Java Virtual Machine (JVM) runs to 3, the number of warmup iterations to 3, and the measurement iterations to 5 to amortize variations in the measurements.[2] We chose this specific configuration because it has yielded stable results on all four systems. The final execution time of a benchmark configuration for a given system is then defined by the median of all 15 measurements.

C. Results & Discussion

Since we evaluated our TFS for four scenarios on four multi-core systems with three different throughput prediction algorithms and various different throughput boundary values, we are not able to present all results in detail. Thus, we first discuss an aggregated view on the results for each benchmark configuration on each multi-core system in Section VI-C1. Afterwards, we exemplarily discuss the results for the *INTEL* system concerning feasibility (see Section VI-C2) and performance (see Section VI-C3). We chose the *INTEL* system since it is the most recent system of all used systems. The benchmarks on the other systems yield similar results. For detailed results, we refer to our replication package [11].

1) Overview: Table II shows the lowest mean execution times of each benchmark configuration for each system which we have measured with and, respectively, without our TFS. The results consistently indicate a speedup for all benchmark configurations on all systems. Hence, our TFS successfully increases the overall throughput. Moreover, the results show that the lowest execution time depends on the used throughput boundary. Different benchmark configurations and different systems require different boundary values. Finally, the lowest execution time also depends on the used prediction algorithm. However, the regression algorithm yields the best results for all benchmark configurations on all systems.

2) Goal 1 (Feasibility): Figure 9 illustrates the total throughput at any point in time while running B1 (balanced workload) on the *INTEL* system with various throughput boundaries.[3] If we consider the time axis, we see that the fastest run of the benchmark takes about six seconds for a throughput boundary below 0.025. All runs with a throughput boundary greater than 0.025 take more than ten seconds.[4] The maximum peak of the throughput is reached at the third second and is kept until the end of the run. Compared to an execution with a higher throughput boundary, e.g., 0.15, the throughput is 30 times higher. These measurements perfectly correlate to the number of stages at the corresponding points in time (see Figure 10). The higher the total throughput, the higher is the number of stages within our TFS.

Hence, our TFS increases the overall throughput (Goal 1a). Furthermore, our SAM automatically adapts the number of stages according to the current runtime workload (Goal 1b).

3) Goal 2 (Performance): Figure 11a to Figure 11d illustrate the execution times of all four benchmarks depending on the used prediction algorithm and the used throughput boundary. Each figure shows that just using the TFS yields to equal or lower execution times independent of the used prediction algorithm. Since the TFS introduces at least two new threads—one for the initial worker stage and one for the merger—this common speedup is comprehensible. Nevertheless, the prediction algorithms differ among each other in a significant way concerning the execution time. In short, the regression algorithm outperforms its alternatives in all four scenarios provided we chose a properly set throughput boundary. Consistent with the results shown in Table II, Figure 11 shows that the lowest execution time on the *INTEL* system is reached by a throughput boundary below 0.025.

[1] http://www.oracle.com/technetwork/java/javase/overview/index.html

[2] The JVM may use different just-in-time compile strategies from run to run. Furthermore, JVM classes are compiled not until they are accessed the first time. Finally, they are optimized for performance on frequent access.

[3] We increased the execution time from 3 sec. to 6 sec. for clarity.

[4] We omitted boundaries greater than 0.15 for clarity.

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

TABLE I: Multi-core systems used for the evaluation

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 II: 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.

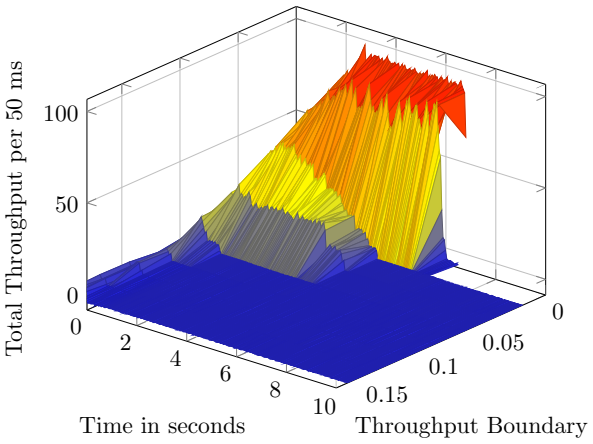


Fig. 9: Total throughput at any point in time while running B1 (balanced workload) on the *INTEL* system with various throughput boundaries

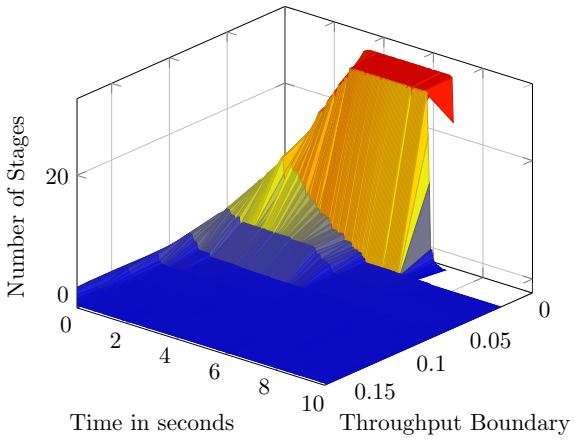


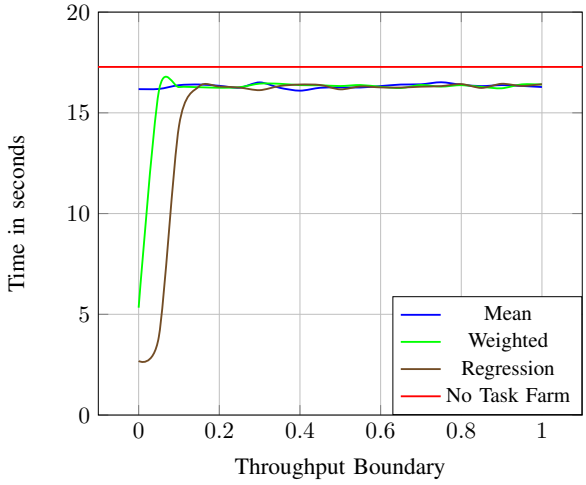
Fig. 10: Total number of stages at particular points in time while running B1 (balanced workload) on the *INTEL* system with various throughput boundaries

Hence, the right choice of the prediction algorithm is crucial for a high overall throughput (Goal 2a). In our benchmarks, the regression algorithm consistently performs best. Moreover, the throughput boundary is also important for a high overall throughput (Goal 2b). In our benchmarks, a throughput boundary between 0 and 0.225 performs best.

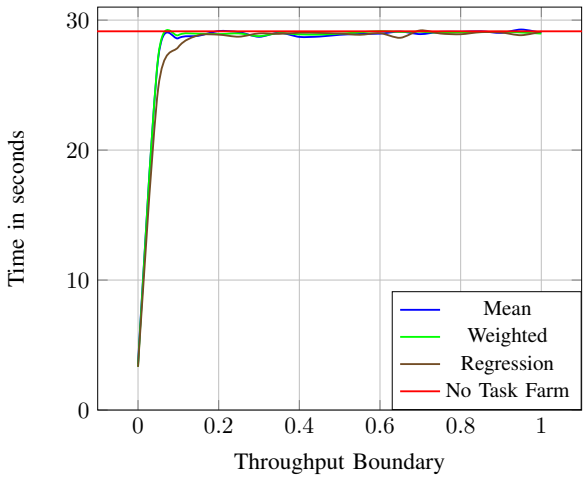
D. Threats to Validity

Internal validity: Although we build benchmarks that represent the four scenarios, we see potential for improvements in the design of the benchmarks. For example, the benchmarks could be changed to implement more realistic use cases. Furthermore, the experimental results depend on our proposed design and the TeeTime-based implementation of both the task farm stage and the self-adaptation manager. Different designs and implementations could lead to different results.

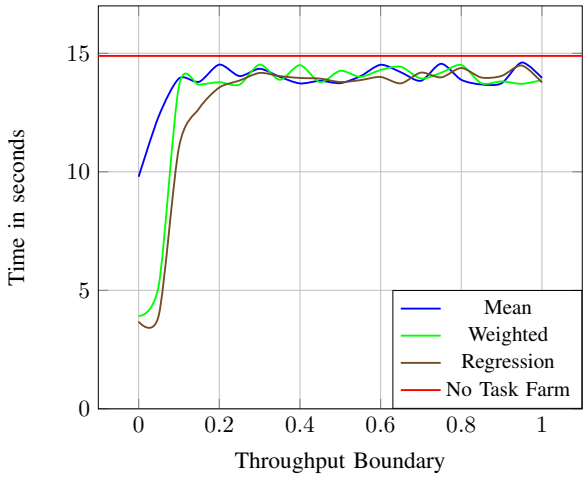
External validity: We evaluated our TFS for four coarse-grained scenarios only. Further scenarios would increase the external validity. Moreover, we evaluated our TFS only on four multi-core systems with different hard drives. Other systems could perform differently and might require an adjustment of the VM/warmup/real iterations to get stable results. Furthermore, our results base on a particular JVM and OS version.



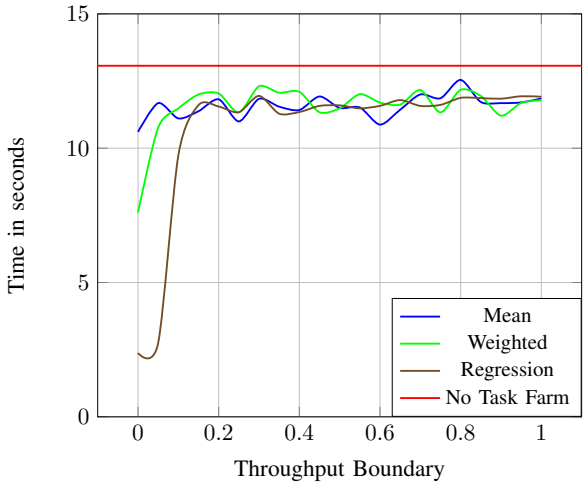
(a) Exec. times of B1 (balanced workload) for the *INTEL* system



(b) Exec. times of B1 (unbalanced workload) for the *INTEL* system



(c) Exec. times of B2 (balanced workload) for the *INTEL* system



(d) Exec. times of B3 (balanced workload) for the *INTEL* system

Fig. 11: Execution times of the benchmarks on the *INTEL* system either with the mean algorithm (blue), with the weighted algorithm (green), with the regression algorithm (brown), or without our task farm stage (red).

VII. Related Work

Some approaches prepare and optimize existing P&F architectures for contemporary distributed systems or for multi-core systems. Probably the best known work is FastFlow [6], [17], a framework for C++-based streaming applications. Similar to the P&F framework TeeTime, it also supports the task farm parallelization pattern. However, the task farm in FastFlow does not support any self-adaptive behavior. A fixed number of worker stages has to be specified at task farm creation. The task farm then uses exactly the specified amount of worker stages for the whole execution. Other P&F-like frameworks, such as Pipes [22] and Akka [23], do not provide both a self-adaptation mechanism and a task farm.

There are also other patterns similar to the task farm parallelization pattern. For example, the Map-Reduce pattern [24] is also able to process multiple tasks concurrently. It uses a *map*-, a *shuffle*-, and a *reduce*-function to efficiently process unstructured data, such as text files, in a distributed manner.

The Fork-Join parallelization pattern [25] is another alternative to the task farm pattern. It can also be applied to parallelize stages on demand by forking and joining their executions. However, when using a P&F architecture, we operate on a stream of tasks which could lead to a performance degrade due to the high amount of fork and join operations.

Kephart and Chess [26] propose a reference architecture for self-adaptive software systems which implements the well-known MAPE-K control loop. Van Hoorn [27] reviews and discusses approaches concerning a self-adaptive capacity management for component-based software systems. Moreover, he proposes a MAPE-K-based framework for architecture-based online capacity management called *SLAstic*. Related MAPE-K frameworks are Rainbow [28], AQuA [29], and the Adaptive Server Framework [30]. We adopt and specialized parts of the

common MAPE-K control loop and the SLAstic [31] meta-model to structure our self-adaptation manager.

StreamIT [32] is a language for building and executing high-performance streaming applications. Similar to our task farm stage, it provides a so-called split-and-join filter which is able to automatically parallelize the underlying filter on demand.

Suleman et al. [9] use a parallelization approach which also adapts the number of worker stages in pipelines according to the workload. They apply an initial training phase in which the stages are monitored to determine when a worker stage should be added or removed. However, this training phase is performed only once at the beginning and thus can produce inaccurate reconfigurations for unexpected workloads.

Sugerman et al. [8] present the GRAMPS programming model for graphics pipelines which defines two types of stages: thread and shader stages. Thread stages are stateful and must be manually instantiated. Shader stages are stateless and automatically instantiated by the scheduler. Sanchez et al. [33] present a GRAMPS-scheduler that performs fine-grained dynamic load balancing and automatic parallelization by scaling such shader stages on demand. Our approach is able to scale stateless and stateful stages within the limits of the `ITaskFarmDuplicable` interface.

VIII. Conclusion

So far, concurrency in P&F architectures is often handled, if at all, only at a coarse-grained level for distributed cluster systems, neglecting parallelization potential of contemporary multi-core processor systems. Parallelizing each filter is not very effective, especially when (1) the workload is unevenly distributed among all filters and when (2) the number of available processing units exceeds the number of filters.

Hence, we propose a composite filter which implements the task farm parallelization pattern. This filter is able to parallelize the underlying filter in order to increase the throughput. Furthermore, we integrate a self-adaptation manager. It allows to automatically adapt the number of filter instances in order to achieve a high throughput even under an unevenly distributed workload. We showed by an extensive experimental evaluation that (1) our task farm filter is able to increase the throughput of various P&F architectures and (2) that our self-adaptation manager scales well when using the regression algorithm for throughput forecasting. We achieved speedups ranging from 1.5 to 7.3 for our benchmarks.

As future work, we plan to identify the best throughput boundary in an automatic fashion at runtime. In this way, the programmer does not need to manually define and adjust it anymore. Moreover, we plan to support duplicable stages with more than one input port and more than one output port.

References

[1] M. Shaw, "Larger scale systems require higher-level abstractions," *SIGSOFT Softw. Eng. Notes*, vol. 14, no. 3, 1989.

[2] G. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture," in *Proc. of the 1st FSE*, 1993.

[3] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

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

[5] 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.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, Oct. 2013, ch. 13.

[7] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. of the 12th International Conference on ASPLOS*, 2006.

[8] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "Gramps: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, 2009.

[9] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *Proc. of the Int. Conf. on PACT*, 2010.

[10] C. Wulf, N. C. Ehmke, and W. Hasselbring, "Toward a generic and concurrency-aware pipes & filters framework," in *Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days*, Nov. 2014.

[11] C. Wulf, C. C. Wiechmann, and W. Hasselbring, "Data for: Increasing the Throughput of Pipe-and-Filter Architectures by Integrating the Task Farm Parallelization Pattern," Mar. 2016, doi: 10.5281/zenodo.46776.

[12] R. Allen and D. Garlan, "Formalizing architectural connection," in *Proc. of the 16th ICSE*. IEEE Computer Society Press, 1994.

[13] S. S. Gokhale and S. M. Yacoub, "Reliability analysis of pipe and filter architecture style." in *the Proc. of the 18th SEKE*, 2006.

[14] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-conditioned, Scalable Internet Services," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, Oct. 2001.

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

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

[17] "FastFlow Documentation." [Online]. Available: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:about

[18] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Addison-Wesley Professional, 2004.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[20] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis," in *Proc. of the ICPE*, 2012.

[21] The TeeTime project. [Online]. Available: http://teetime.sf.net

[22] The TinkerPop project. [Online]. Available: http://www.tinkerpop.com

[23] The Akka Framework. [Online]. Available: http://akka.io

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Comm. of the ACM*, vol. 51, no. 1. ACM, 2008.

[25] D. Lea, "A Java fork/join framework," in *Proc. of the ACM Java Grande 2000 Conference*. New York, NY, USA: ACM, 2000.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] A. van Hoorn, M. Rohr, I. A. Gul, and W. Hasselbring, "An adaptation framework enabling resource-efficient operation of software systems," in *Proc. of the Warm Up Workshop for the ICSE*, 2009.

[32] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proc. of the Int. Conf. on CC*, 2002.

[33] D. Sanchez, D. Lo, R. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *Proceedings of the 20th International Conference on PACT*, Oct. 2011.