# Towards Energy Auto-Tuning

Sebastian Götz, Claas Wilke, Matthias Schmidt, Sebastian Cech and Uwe Assmann
sebastian.goetz@acm.org, {claas.wilke, matthias.schmidt, sebastian.cech, uwe.assmann}@tu-dresden.de
Technische Universität Dresden, Department of Computer Science, Software Technology Group, D-01062 Dresden

***Abstract*—Energy efficiency is gaining more and more importance, since well-known ecological reasons lead to rising energy costs. In consequence, energy consumption is now also an important economical criterion. Energy consumption of single hardware resources has been thoroughly optimized for years. Now software becomes the major target of energy optimization. In this paper we introduce an approach called energy auto-tuning (EAT), which optimizes energy efficiency of software systems running on multiple resources. The optimization of more than one resource leads to higher energy savings, because communication costs can be taken into account. E.g., if two components run on the same resource, the communication costs are likely to be less, compared to be running on different resources. The best results can be achieved in heterogeneous environments as different resource characteristics enlarge the synergy effects gainable by our optimization technique. EAT software systems derive all possible distributions of themselves on a given set of hardware resources and reconfigure themselves to achieve the lowest energy consumption possible at any time. In this paper we describe our software architecture to implement EAT.**

## I. Introduction

The energy use of servers is steadily raising and soon will pass the asset costs as the U.S. Environmental Protection Agency (EPA) shows in their report on server and data center energy efficiency [1]. According to this report, the energy consumption of servers doubled from year 2000 to 2006 and will double again until 2011. More than 100 billion kWh (approx. $7.4 billion) will be the annual electricity consumption in the U.S. in 2011. The EPA recommends research and development activities to improve the energy efficiency of servers and also points out the necessity to investigate potential savings by power management across multiple resources [1, p. 118].

The energy used by hardware resources should be proportional to their utility for end users. In other words, if the end user does not utilize resources by using software running on top of them, the resources should not use any energy. This issue is known under the term of *energy proportionality* [2]. Recent work shows that we are far from energy proportionality. In [3] Tsirogiannis et. al. reveal that over 50% of the overall power consumption is caused by servers with idle load. Moreover, they detected that resource utilization does not directly correlate to its energy use. The actual energy use depends on the kind of task the resource has to accomplish. They show a 60% variation of power consumption for the same level of resource utilization. These results substantiate the nonexistence of energy proportionality. (On the other hand they show, that energy is proportional to performance, having idle power as offsets). Tsirogiannis et. al. point out the optimization of multiple, jointly used resources as a promising direction, too [3, p. 242].

Several problems derive from the energy-unawareness of IT infrastructures used to run distributed software. First, the energy consumption of software components is hard to predict. This is, because energy consumption of software components depends on the hardware they are running on and the user that interacts with the components. The users demand as well as the users utility w.r.t. service requests need to be considered. Energy efficiency is the balance between user utility and energy consumption. However, resource usage by software components and the user's workload are not explicitly taken into account in current software development processes. Hence, to predict energy consumption of software components and correlate it with user's requirements, an energy-aware software architecture and runtime environment are required.

In this paper we introduce an approach for *energy auto-tuning (EAT)* software systems. Such systems derive all possible distributions of their software on a given set of hardware resources and reconfigure themselves to achieve the lowest energy consumption possible at any time. Our focus are distributed, component-based applications. We therefore propose the *Cool Component Model (CCM)* together with the *Energy Contract Language (ECL)* as appropriate means to capture energy-aware software architectures. Furthermore, we propose the *THEATRE* as our energy-aware runtime environment. We thus also investigate a development process for energy-aware software systems and do not solely focus on energy auto-tuning. This is, because energy auto-tuning requires such an energy-aware software architecture.

The rest of this paper is structured as follows. In Section II we introduce a running example. Afterwards, we highlight requirements for EAT systems in Section III. Our proposed software architecture and runtime environment are presented in Section IV. Finally, we outline related work in Section V and conclude and give pointers for future work in Section VI.

## II. Video Server Example

In this section we present an example of a small component-based system that can be energy-optimized using EAT. The example was adapted from a case study depicted in [4]. It describes a simple video server scenario consisting of two software components: a `VideoServer` and a `VideoPlayer` (cf. Fig. 1). The `VideoServer` is located at a server whereas the `VideoPlayer` is deployed at clients. The `VideoServer` provides services to select and transmit videos that are stored on a `FileServer`. The `VideoPlayer` can receive video streams via a network

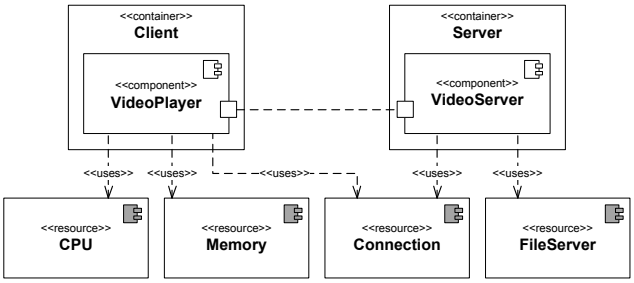


Fig. 1. The components of the VideoServer example; cf. [4].

`Connection`, decodes the video frames and displays them on the screen.

Besides software components, hardware resources are involved in the presentation of a video. First, the `VideoServer` requires a `FileServer` to select and deliver file streams to clients. Furthermore, a network `Connection` is required. On the client side, `CPU` and `Memory` are used to decode and display the video. One may argue, that further resources like a monitor and speakers are required to present the video. Anyhow, the example was designed to illustrate the concepts and thus, remained as simple as possible.

Different clients require to display videos in different qualities, e.g., a mobile phone does not require the same resolution as a desktop PC. Hence, the `VideoServer` and the `VideoPlayer` provide their services in different qualities that can differ in resolution and frame rate. Resources can provide different quality profiles for their services as well. E.g., a `Connection` can provide different bandwidths, or a `Memory` unit can provide different amounts of space. Using these quality profiles in combination with component-to-component and component-to-resource dependencies allows to switch between different configurations of the system at runtime. E.g., the application could use a low-quality resolution if a video was requested from a mobile phone to save bandwidth, CPU and, most importantly to us, energy.

## III. Requirements for EAT Systems

After giving a running example we now highlight requirements for an EAT system, which can be classified into component modeling facilities, expressing hardware/software dependencies and requirements regarding an energy-aware runtime environment. These requirements are the basis for our main contributions, namely the CCM component model, the ECL contract language and the energy-aware runtime environment THEATRE.

### A. Modeling of Software and Hardware Components

An EAT system requires a special software architecture, whose building blocks are explicitly connected to resources. With resources, we do not just denote hardware resources, like a central processing unit, but virtual resources, like operating systems and files, too. We propose *components* [5] to describe both, software and hardware elements of an EAT architecture. Furthermore, our proposed component model CCM has to fulfill the following requirements:

*1) HW-SW modeling:* Software only consumes energy in an indirect way by using hardware resources where the software components run, i.e. the hardware consumes energy by executing software components. In order to increase energy efficiency of IT infrastructures, it is not sufficient to capture the architecture of software components. It is also necessary to take hardware resources of the infrastructure into account.

*2) Quality-of-Service (QoS) properties:* The component model should also provide facilities to express quality of service properties of hardware and software components (e.g., energy consumption of a CPU). Such QoS properties are the basis for optimizing energy efficiency at runtime.

*3) Variant modeling:* Each component of an EAT architecture can exist in different implementation variants with a common interface. Offered services from the component can differ in quality and therefore in the energy consumption of their underlying hardware. Hence, the CCM has to provide concepts to express implementation variants of software components including implementation specific QoS properties.

*4) Behavior modeling:* Hardware components are able to reside in different performance states (e.g., as defined in the ACPI specification [6]). Each performance state implies a specific energy consumption of hardware resources. Such performance states as well as the underlying behavior should be regarded in a suitable component model. Behavior modeling of software components is also necessary because it implies which hardware resources are used by a component.

### B. Modeling of Software/Hardware Dependencies

Besides defining central building blocks of software, a software architecture defines, how these building blocks are connected to each other. Due to the different variants of provided and required services we propose to use contracts as connectors for components. They can be realized by our contract language ECL, which has the following requirements:

*1) Dependencies between components:* The contract language should be able to express dependencies between different software components and between software components and required resources.

*2) Energy Consumption:* The contract language should allow to define energy consumption of resources according to their current state. The energy consumption of software components should be computable via their component and resource dependencies. Energy consumption, which is not caused by running software on resources, e.g. of resources in idle mode, is described in the behavior part of the component model.

*3) Quality Modeling:* Besides energy consumption, the contract language should support other functional and/or non-functional requirements for components and resources (e.g., qualities like frame rates, response times and resolutions).

*4) Variant modeling:* Similar to the component model the contract language should allow to describe different states for components that lead to different QoS for provided and required services.

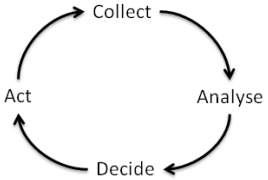


Fig. 2. Autonomic Control Loop of auto-tuning Systems, cf. [7]

### C. Energy Auto-Tuning Runtime Environment

Components, with explicitly defined resource usage, do not improve energy efficiency on their own. The problem is, that there is no energy-aware authority that manages requests of software on hardware. Such an authority is our proposed *THree-layer Energy auto-tuning Runtime Environment (THEATRE)*. To optimize the efficiency of software systems w.r.t. user requests and energy consumption the authority needs to enforce the best system configuration for current and forthcoming user requests. A system configuration describes which component implementations are being used and at which resources the corresponding instances are deployed. Therefore, THEATRE has to fulfill four requirements conforming to the autonomic control loop principle of auto-tuning systems [7], which is depicted in Fig. 2.

*1) Collect:* The EAT system should be aware of the existing software components and hardware resources (either directly or indirectly via hierarchies of sub-systems).

*2) Analyze:* For every user request, the EAT system should be able to analyze the request according to its required components, resources and, in consequence, its energy consumption. All possible deployments of the required components should be computed and evaluated w.r.t. their energy consumption.

*3) Decide:* The EAT system should select the energy-optimal configuration for current and future service requests.

*4) Act:* Finally, the EAT system must be able to reconfigure the system according to the energy-optimal configuration computed before.

## IV. Energy Auto-Tuning Systems

This section elaborates on our energy-aware software architecture, which consists of our proposed component model and contract language, and our energy auto-tuning runtime environment (THEATRE).

### A. An Energy-aware Software Architecture

According to the requirements presented above, our proposed software architecture consists of components and contracts, which are described in the following.

*1) Components:* To model the parts of an IT infrastructure (user, hardware resources and software components) we designed the *Cool Component Model (CCM)*. CCM extends and combines ideas of the SPEEDS meta model [8] and the MADAM component model [9] to allow modeling of a system's structure, behavior and energy consumption. Besides these three aspects, defining variations is a major concern in CCM. As the definition of energy consumption will be discussed in Section IV-A2, this section will focus on structure, behavior and variation modeling. For each of these three aspects our component model has an own package. Nevertheless, these packages cannot be seen as distinct parts, but are interconnected.

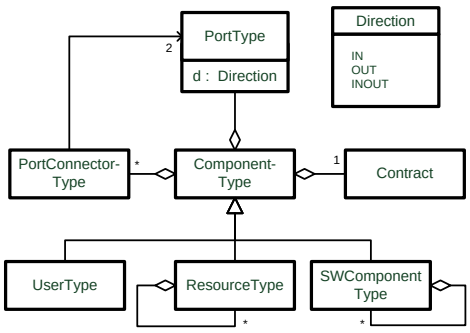


Fig. 3. The structure package of CCM

To model the structure of a system CCM provides concepts which are grouped in the *structure package* illustrated in Fig. 3. This package can be used to define a hierarchical IT infrastructure, it's interfaces, interactions between interfaces as well as contracts that are valid between them. Furthermore, it acts as a type system for architectural concepts in CCM. These types can be used later on to define variations of specific parts of the system to allow energy auto-tuning.

A central concept in CCM are components, which are represented by `ComponentTypes` in the structure package. `UserType`, `ResourceType` and `SWComponentType` are specific components referring the different parts of an IT infrastructure, where the latter two concepts form a hierarchy using self containments. Furthermore, `ComponentTypes` contain a number of `PortTypes`, which define the component's external interface. A `PortType` contains a `Direction` to specify whether the component offers (`OUT`) or requires (`IN`) a service via this port or if it does both (`INOUT`). `PortConnectorTypes` enable components to combine (sub)components via their ports and model interactions in this way. Finally a `ComponentType` has a `Contract` that specifies what the component requires and provides.

To model behavior, CCM provides concepts which are contained in the *behavior package* presented in Fig. 4. This package can be used to define a behavior template of a component and workloads that invoke this behavior at certain points in time. Modeling the behavior is of special importance to the components and resource simulation (see Sec. IV-B) as it allows to define energy consumption over time. The central concept of the behavior package is the `BehaviorTemplate`. It defines the behavior of a component and contains `CostParameters` which need to be substituted by a concrete variant of the component. For instance, a `BehaviorTemplate` can be a `StateMachine` while states and transitions might contain energy consumption or time as `CostParameters`. Moreover, additional subclasses of `BehaviorTemplates` like heuristics are possible. Each

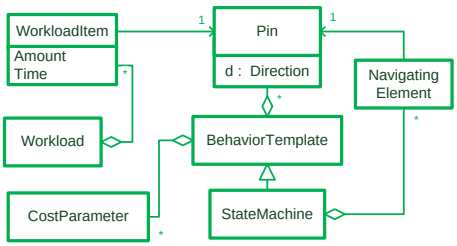


Fig. 4. The behavior package of CCM

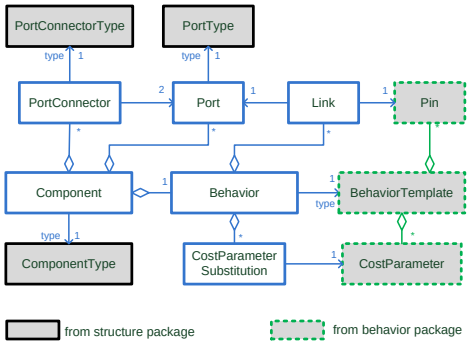


Fig. 5. The variation package of CCM

`BehaviorTemplate` contains `Pins` which are equivalent to `Ports` in the structure package. A `Pin` defines the external interface of a `BehaviorTemplate` and is connected via `NavigatingElements` to parts of the `StateMachine`. Later on, a concrete variant of a component connects a pin with a port and that way can model what happens if the service of a port gets invoked.

To model variation, CCM provides concepts which are contained in the *variation package* depicted in Fig. 5. This package can be used to define concrete variants of components by connecting one `ComponentType` to one `BehaviorTemplate` and substituting its parameters. Furthermore, it allows to define variants of complete (sub)systems which can be used by THEATRE to decide which system configuration fits to the user's needs and in that way is the most energy efficient. Central to the variation package are the concepts `Component` and `Behavior`. A `Component` refers to a `ComponentType` and can be specialized in the same way (User, Resource, SWComponent - not depicted in Fig. 5). Furthermore, it contains `Ports` and `PortConnectors`, which refer to their types respectively. These two concepts and the ability to define sub-variants allow to specify complex variations of complete systems. The `Behavior` refers to a `BehaviorTemplate` and substitutes its cost parameters using `CostParameterSubstitutions`. Finally, a `Behavior` owns `Links` that connect `Ports` of the component variant to `Pins` of the `BehaviorTemplate`.

Given the three packages *structure*, *behavior* and *variation* CCM provides advantages regarding the following three aspects. First, it allows to specify the architecture of software components and resources and hence to build a repository of IT infrastructure parts. Second, it allows to define behavior independently of a component's structure and in that way increases reuse. Third, CCM provides concepts to model alternatives, which enables THEATRE to find the most energy efficient system configuration.

*2) Contracts:* To describe and compute the energy consumption of software components, we use the *Energy Contract Language (ECL)*. ECL is an advancement of the *Component Quality Modeling Language (CQML)* [10] and *CQML+* [11] w.r.t. energy consumption. ECL allows to define QoS-contracts using characteristics, qualities and profiles for resources and software components. ECL contains the description of the dependencies between software components and resources and the variability of these compositions.

Characteristics are measuring units used to express qualities in profiles. E.g., the `frameRate` of a `VideoStream` can be specified as shown below, defining the type of `frameRate` and how its values are derived from `VideoStreams`.

```
1  characteristic frameRate(stream:VideoStream) {
2    domain numeric integer frames/second;
3    value stream.getFrameRate();
4  }
```

The major concept of ECL are profiles that specify the different states of resources and software components. E.g., the resource `Network` has two different states which are `connected` and `disconnected`:

```
1  profile NetworkStates for Network {
2    state connected {
3      energy-effect 0.8 Watt;
4      provides characteristic bandwidth = 600;
5    }
6    state disconnected {
7      energy-effect 0.4 Watt;
8    }
9    transition connected -> disconnected {
10     whenever event-occurs disconnect();
11     delay 1.0 seconds;
12     energy-effect 5.0 Watt;
13   }
14   transition disconnected -> connected {
15     whenever event-occurs send();
16     delay 10.0 seconds;
17     energy-effect 30.0 Watt;
18   }
19   initial-state disconnected;
20 }
```

The different profile states contain an `energy-effect` that specifies, how much energy is consumed for a resource being in this state. Optionally, a state can provide characteristics for software components using this resources, such as the `bandwidth` provided in the `connected` state. Transitions between profile states can be defined specifying their duration and energy-cost. Events declare when such transitions should be performed. Finally, profiles for resources have to specify an `initial-state` for their instances.

Besides profiles for resources, profiles have to be defined for software components as well. Below, a profile for the `VideoServer` component introduced in Sec. II is shown:

```
1  profile ServerProfile for VideoServer {
2    state highQuality {
3      uses characteristic bandwidth = 450;
4      provides characteristic frameRate = 30
5        and characteristic imageWidth = 352
6        and characteristic imageHeight = 288;
7    }
8    state lowQuality {
9      uses characteristic bandwidth = 150;
10     provides characteristic frameRate = 10
11       and characteristic imageWidth = 176
12       and characteristic imageHeight = 144;
13   }
14   transition any-state -> any-state {}
15   precedence highQuality, lowQuality;
16 }
```

As specified, the `VideoServer` provides two different states, these are a `highQuality` and a `lowQuality` state. The states differ in the resolution (`imageWidth` and `imageHeight`) and `frameRate` of their provided video stream. Besides the provided resolution and frame rate, both states require a characteristic `bandwidth` provided by `Network` resources. But as expected, the `highQuality` requires a higher amount of `bandwidth` than the `lowQuality` state. The `VideoServer` profile is now used as a basis for a profile of the `VideoPlayer` introduced in Section II:

```
1  profile PlayerProfile for VideoPlayer {
2    state highQuality {
3      uses profile ServerProfile::highQuality
4        and characteristic bandwidth = 450
5        and characteristic cpu_usage = 66.7
6        and characteristic ram = 200.0;
7      provides characteristic frameRate = 30
8        and characteristic image_width = 352
9        and characteristic image_height = 288;
10   }
11   state lowQuality {
12     uses profile ServerProfile::lowQuality
13       and characteristic bandwidth = 150
14       and characteristic cpu_usage = 25.0
15       and characteristic ram = 50.0;
16     provides characteristic frameRate = 10
17       and characteristic image_width = 176
18       and characteristic image_height = 144;
19     resources
20   }
21   transition any-state -> any-state
22 }
```

Like the `VideoServer`, the `VideoPlayer` provides two different states `highQuality` and `lowQuality`. As illustrated above, the states provide the same qualities as the `VideoServer`, but require a specific state of the `VideoServer` to provide their own service. This dependency information is required to specify, which components can be deployed and composed to provide specific QoS at runtime. Required resources can be reserved; the resource profiles can be used to predict the energy consumption of a specific provided software service. Currently, ECL is still in an early development state. Initial tool support, including a text editor with editing support like syntax highlighting and code completion, has been realized using *EMFText* [12].

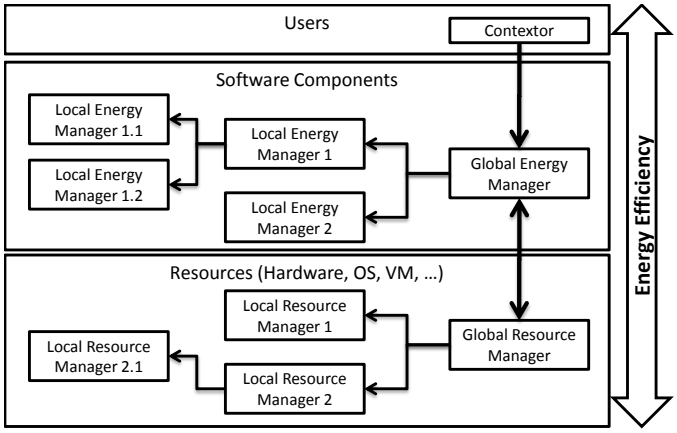


Fig. 6. General Architecture of THEATRE - **TH**ree layer **E**nergy **A**uto **T**uning **R**untime **E**nvironment

### *B. THEATRE - A Runtime Environment for Energy Efficient Distributed, Component-based Software Systems*

Using software components and contracts to design and implement software systems does not lead to better energy efficiency on its own. Instead the knowledge about the different system variants, along with their cost and utility, needs to be used by a runtime environment, which is able to reconfigure the running system. The decision, which variant is best, depends on the user's needs, which vary over time. In consequence, the best alternative is known only at runtime and is likely to change over time. A system variant denotes a mapping of specified component implementations onto resources.

We propose to use a runtime environment, which consists of three layers: a *user*, *software* and *resource layer* as depicted in Fig. 6. On the user layer *contextors* are used, to collect data about the user's needs. Analogously, on the resource layer, *resource managers* are used, to collect information about the energy behavior of resources. It is important to note that resources are not just hardware resources, but complex, virtual resources, like the operating system or files. The centered layer comprises software components and *energy managers*, which use the knowledge of contextors, resource managers and their own knowledge about existing components and their variations (implementations), to choose the currently most energy efficient system variant. If the energy manager decides on a system variant, which is not the current, it forces a reconfiguration. This includes the migration of components from one resource to another. In consequence, resource managers need to provide functionality to steer resources. For example, when a resource is unused, after the last software component has migrated somewhere else, it should be powered down in order to save energy. In the following, each layer is examined in more detail.

*1) User layer:* To reason about the user's utility, the expectations of the user need to be collected by so-called *contextors*. User utility can be quantified using *user metrics* [13] – human-perceptible qualities of software, like response time or resolution. The easiest way to collect information about the user's

expectations is by letting the user define her expectations explicitly. A well known example is the 'high quality' button of the YouTube video player, which allows the user to state that she expects to see a video in high quality. More sophisticated approaches, like using an ontology to define user-metrics and probabilistic or heuristic approaches to predict the user's expectations can be used as well. Additionally, the user's demand need to be predicted or collected by the contextor. This is, because the demand determines which software components are required.

*2) Resource layer:* The resource layer comprises hierarchical interconnected resources, in which each has a dedicated *resource manager*. This manager has to fulfill five requirements. First, it needs to know about the infrastructure. That is, which other resources can be reached. Second, it needs to know about its resource's properties: What services does the resource offer? What energy states does the resource have? Third, it needs to be able to predict the energy behavior for a workload, which is a timed sequence of service requests. We use a simulation approach, based on *energy state charts* (ESC) [14], to realize the prediction. The central feature of ESCs is, that states have a defined power consumption rate and transitions have defined delays as well as energy costs. Various other approaches exist. Flinn et. al. use models of linear functions [15] and Fei et. al. use heuristics [16]. Models of linear functions and heuristics only provide a possibly good prediction, whereas ESCs allow for predictions, which are as precise as the parameters provided to the charts. Therefore we decided to use ESCs. Last, the resource manager needs to be able to steer the resource and force it into some other energy state. At least, it should offer to switch off the resource and to switch it on again. We distinguish between a *global resource manager* and *local resource manager*s. Whereas local resource managers are tied to their resource, the global resource manager abstracts the whole infrastructure. In [17] we elaborate on resource managers.

*3) Software layer:* The software layer comprises software components and *energy managers*, which reason about valid system variants of the software. In other words, the energy manager assesses mappings of component implementations to resources with energy usage as cost and the fulfillment of user expectations as utility. The system variant with the best combination of cost and utility will be chosen by the manager, which then forces a reconfiguration of the system, if necessary. We distinguish between a *global energy manager* and *local energy manager*s. The local managers are tied to software components and know about their composability by using the ECL contracts specified during software development. The global energy manager knows the local ones and uses their information about valid mappings (system variants) to combine it with energy usage information from the resource layer and utility information from the user layer. It directly communicates with the global resource manager. If a reconfiguration was forced, the utilization of resources changes and might fall below or above thresholds. In consequence, these resources should be forced into another energy state, like a sleep or special performance mode. It is important, that the global energy manager considers the energy required to process the reconfiguration. This is, because reconfigurations swiftly become very energy consuming, for example due to the need to initialize new resources.

In conclusion, information of all three layers is used by the global energy manager which steadily initiates reconfigurations of the system to keep the highest possible ratio of utility and cost (i.e. energy efficiency) at any time.

## V. Related Work

In this section we outline related work from three research areas, which we combined in our overall approach. These areas are: auto-tuning techniques, approaches for energy efficiency in general and quality-aware component models.

### A. Auto-Tuning

*auto-tuning* is a well established optimization technique in the *high performance computing (HPC)* community. Running numerical algorithms (e.g., for matrix multiplication or Fourier transform) near to the peak performance of parallel computers is an important goal in HPC. The performance of a numerical algorithm's implementation depends strongly on its underlying hardware platform as well as the problem size (e.g., matrix size). Therefore, platform specific as well as problem specific optimizations are necessary for such algorithms.

During recent years, several successful numerical libraries with auto-tuning facilities have been developed [18], [19], [20], [21], [22]. *auto-tuning capability* means that libraries are able to adapt themselves to specific hardware architectures. Approaches behind these libraries can be distinguished between optimization at installation time and optimization at runtime. Libraries with optimization facilities at installation time (e.g., *ATLAS* [18], *PhiPac* [21], *Spiral* [22]) analyze properties of a target platform to generate highly optimized code. For instance, in ATLAS analyses are realized based on certain hardware parameters (e.g., capacity of L1 data cache, number of registers) to determine additional parameters like loop unrolling factors for the following code generation phase. Generated code is then executed for benchmarking purposes. Achieved MFLOPS are traced back into the analysis component of ATLAS. Other auto-tuning libraries like OSKI [20] optimize not only the implementation code but provide also optimized data structures. Optimization at runtime is for instance provided by FFTW [19], which can be used for Fourier transform. FFTW provides a set of optimized composable code blocks (codelets) performing a specific part of transform. An optimized transform is created by composing several codelets to a so called plan. Internally plan generation is realized by measuring the performance of available plans and selecting the fastest implementation. For executing a plan FFTW provides an separate executor.

The commonality behind the auto-tuning approaches mentioned above is the autonomic control loop principle [7]. As explained in Section III we consider this principle as a