Global and Thread-Local Activation of Contextual Program Execution Environments

Markus Raab
Vienna University of Technology
Institute of Computer Languages, Austria
Email: markus.raab@complang.tuwien.ac.at

***Abstract*—Ubiquitous computing often demands applications to be both customizable and context-aware: Users expect smart devices to adapt to the context and respect their preferences. Currently, these features are not well-supported in a multi-core embedded setup. The aim of this paper is to describe a tool that supports both thread-local and global context-awareness. The tool is based on code generation using a simple specification language and a library that persists the customizations. In a case study and benchmark we evaluate a web server application on embedded hardware. Our web server application uses contexts to represent user sessions, language settings, and sensor states. The results show that the tool has minimal overhead, is well-suited for ubiquitous computing, and takes full advantage of multi-core processors.**

I. Introduction

Context-awareness aims to give users the impression of devices to be smart. We want devices to react according to properties of their physical environments. E.g., when a mobile phone does not feel body temperature anymore, it can deactivate vibration because the user would not sense the vibration.

Customizability intends to give users the opportunity to change unwanted defaults, hence bringing the behavior of the device in line with their needs. For example, if users are deaf or blind, the deactivation of vibration might not be appropriate, because they may rely on sensing the vibration.

Multi-Core Processors pose new challenges and are undoubtedly an upcoming trend for embedded computing. Multi-threading is a popular technique to better facilitate multi-core processor resources.

These three issues seem unrelated, but are often found together in ubiquitous computing. Previous work on context-changes did either lack good support for customizability or multi-threading. Changes to context either:

influenced all threads which has severe performance drawbacks because of the needed thread synchronization or

influenced only a single thread which is not expressive enough because the context can affect the whole device.

In this paper, we describe a tool that empowers developers to effortlessly provide different customizations in different contexts for multi-threaded applications. *Program execution environments*, in a broad sense, are the influence of systems on programs, including command-line arguments and configuration files. Our idea is to support activation of such contextual program execution environments selectively in threads.

On the one hand, our tool enables threads to globally change the context when sensors detect an event with consequences for the whole device, e.g., when the battery is low. On the other hand, our tool allows us to have local context changes that deal with local issues, e.g., language settings for a specific Hypertext Transfer Protocol (HTTP) server request.

Our tool provides flexibility for the programmer. Nevertheless, it is safe to use: It takes care of necessary synchronization. It has a build-in way to serialize all variables, which is needed for persistency of, e.g., customization.

Our tool has the potential of increasing efficiency: It decreases development effort because of code generation and it increases performance by caching. Finally, the tool is extensible because the code generator is adaptable.

The paper tackles the following questions:

1) How can context-awareness be semantically more powerful for ubiquitous computing?
2) Is the overhead of our tool, especially in embedded and multi-threaded scenarios, acceptable?

These questions are significant, because embedded computing tends to have multi-core processors more often, while many other resources are still very limited.

The structure of the paper is as follows: In Section II we take a look at the state of the art and our previous contributions regarding context-awareness. In Section III we present our tool. In Section IV we report on a case-study with a web server on embedded hardware. After comparing with related works in Section V, we finally conclude the paper in Section VI.

II. Background

Context-oriented programming (COP) allows us to naturally separate multi-dimensional concerns [1], [2]. It is an extension of object-oriented programming. Its main concept is the activation and deactivation of so called *layers*. Every layer adds an additional dimension that cuts across the context-aware system. Layers can be seen as a modularization concept [3].

The activation and deactivation of layers can happen at any time during program execution. A currently active stack of layers defines the context the program or thread is in. COP permits us to specify programs with very dynamic behavior. In terms of efficiency two main issues exist:

1) The activation and deactivation of layers is expensive.
2) The use of context-aware objects is expensive.

For both issues improved solutions were proposed: Costanza et al. [4] used already optimized language-features for layer activation and Bockisch et al. [5] introduced control flow guards. A comparison [3] revealed that the implementations, relative to host language implementations, are still very inefficient despite these optimizations. Performance penalties of 75% to 99% [3] made COP unattractive for embedded or ubiquitous computing. The improved solutions performed only better in specific cases, e.g., without active layers.

The main reason for the modest performance is that these approaches dynamically adapt application code. Tanter [6] proposed a lightweight alternative: *Contextual values*. They seem to be easier to learn because they "boil down to a trivial generalization of the idea of thread-local values" [6].

Contextual values stem from COP and naturally work along with the concepts of dynamic scoping and layers. They allow us to limit side-effects to an enclosing context. Different from previous approaches, no application code is adapted. Because of that property, we were able to demonstrate [7] that an implementation of contextual values can have zero overhead on access, relative to the host language, despite of active layers.

In the remainder of this section we will give a brief introduction to our previous work [7]. Contextual values are specified in configuration file syntax, e.g., INI:

```
[/watchdog/%security%/enable]
type=boolean
```

Here we specify `enable` as a contextual value of type `boolean` below another contextual value `watchdog`. A tool generates the code for the underlying classes `Enable` and `Watchdog` from the specification. All classes are nested in one large hierarchy, with `Environment` as their root element.

Parts of the specification enclosed in `%`, e.g. `%security%`, are placeholders. A single contextual value can have many values: one value for every context. These values are stored in a key set data structure. The unique keys needed to lookup individual values is determined by substitution of the placeholders.

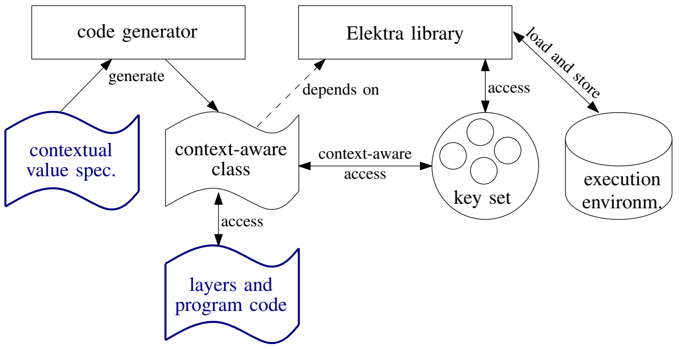

Fig. 1. Overview of Elektra [7], the implementation of our tool. The bold (blue) boxes need to be implemented by the user of the tool.

In Figure 1 we see that the context-aware classes are generated using the contextual value specification. The library part of *Elektra* (i.e. an implementation of our approach [7] written in C), maps the *program execution environments* (i.e., command-line arguments and configuration files) to the in-memory data structure *key set*. Contextual values are initialized using the key set. Additionally, the context-aware classes use the key set to store the concrete values for every context. They respect the context by using a key with all placeholders of the specification substituted. Furthermore, Elektra allows us to remember customizations by serialization of contextual values back to the program execution environments.

To work with Elektra, developers only have to specify the contextual values and layers as shown by the bold (blue) boxes in Figure 1. Then, developers can directly use the contextual values in the own code in the same way as variables are used:

```
void printWatchdogStatus(Watchdog::Enable const & e)
{
    if (e) { cout << "Watchdog is enabled"; }
}
```

Using the methods `with` and `without` the user changes the context, potentially influencing contextual values. In Elektra values are only affected iff one of the placeholders in the specification matches the identification of the layer:

```
void enableWatchdogInSecurityContext(Watchdog::Enable & e)
{
    bool originallyEnabled = e;
    assert(e.getName() == "/watchdog/%/enable");
    e.context().with<Security>("A")([&]{
        e = true; // security context "A" active here
        assert(e.getName() == "/watchdog/A/enable");
    }); // end of security context "A"
    assert(e == originallyEnabled);
    assert(e.getName() == "/watchdog/%/enable");
}
```

For the understanding of this paper the syntactic details of C++11 lambdas are irrelevant. All we have to know is that the first parentheses after `with` contain arguments for layer construction. The block after the capture list `[]` is the lambda function. It will be executed in the same thread, but in another context.

The first `assert` in the function `enableWatchdog InSecurityContext` states that the layer `Security` is inactive, indicated by the `%` as empty layer name. In the next line we activate the layer `Security` using the `with` statement. Within the block, the placeholder `%security%` is replaced with the security level `A`.

From the `assert` at the end we still get the original value, because the layer `Security` is inactive. Nevertheless, the function has a side-effect: `e` is changed in `Security`-context `A`. The resulting configuration in the key set is, e.g.:

```
/watchdog/%/enable=false
/watchdog/A/enable=true
```

Layers are specified in the host language and need to provide a method `id()` that refers to the placeholder. The method `operator()` allows us in C++ to mimic function invocation on objects. The returned string of the function invocation is used to substitute the placeholder. By returning an empty string, layers can pretend that they are inactive. The layer must be implemented by the developer, e.g.:

```
class Security : public kdb::Layer
{
public:
    Security(std::string level) : level(level) {}
    std::string id() const { return "security"; }
    std::string operator()() const { return level; }
private:
    std::string level;
};
```

According to our previous work [7], Elektra has following properties:

1) Zero overhead when reading contextual values.
2) Classes for contextual values are synthesized using code generation from the specification.
3) A unique name for every contextual interpretation eases debugging and enables persistency.
4) Automatic initialization of values from contextual program execution environments is integrated.

III. CoElektra

In this section we will explain an extension of Elektra, called CoElektra, using examples we faced in the development of embedded systems. These include, but are not limited to, power saving, tampering, and different hardware setups. Additionally, we refer to issues of web servers running on such hardware.

A. Activation

Using the methods `activate()` and `deactivate()` we are not limited to a single block but switch the context globally:

```
void enableWatchdog(Watchdog::Enable const & e)
{
    assert(e.getName() == "/watchdog/%/enable");
    e.context().activate<Security>("A");
    assert(e.getName() == "/watchdog/A/enable");
    assert(e == true);
} // Security Layer A stays active
```

While effects of the `with`-statements are bound to a single thread of execution, in CoElektra, `activate()` and `deactivate()` potentially influence every thread of execution:

```
c1.deactivate<BatteryLow>();

// Security unchanged
```

```
c2.activate<Security>(cv);
// BatteryLow inactive
```

The listing shows the parallel execution of two threads. The local variables `c1` and `c2` represent the context and hold the currently active layers per thread. The semantics of (de)`activate()` are as follows: At the end of the left side, the layer `BatteryLow` is inactive and `Security` is unchanged, while on the right side, `BatteryLow` is inactive and `Security` is active. We see that (de)`activate()` respect previous invocations, even if they happened in another thread. The context `c2` updates the layers deactivated by `c1` before it will activate `Security`. This synchronization of layers before (de)activation is an important property: Layers can depend on each other and only this way we can guarantee that the context consists of the same layers with the same state for all threads. In the example above, the contextual value `cv` already considers the inactive layer `BatteryLow` at activation time.

Sometimes we want to update a context, i.e., update the layers changed by other threads, without introducing changes to the global active and inactive layers:

```
c1.activate<BatteryLow>();
```

```
c2.syncLayers();
// BatteryLow active
```

The method `syncLayers()` updates the layers of the context `c2` considering every `activate()` or `deactivate()` of any thread that happened up to that moment. After it is executed, the layers in `c2` will be the same as in the context that caused the last layer activation, i.e. `c1` in the example. In other words: The thread of execution using `syncLayers()` has the same active and inactive layers that it would have had, if it had executed every `activate()` and `deactivate()` of the program itself. Every thread that recently used `syncLayers()` will have the same active layers. Yet, during the execution of a `with`- and `without`-block, individual layers can temporarily differ.

B. Coordination

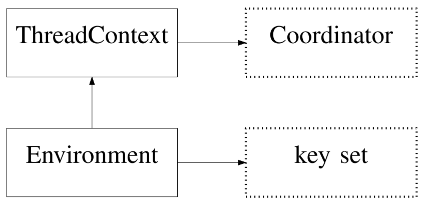

Fig. 2. Coordination between major parts involved. The dotted boxes can be shared between threads. Arrows indicate a composition.

Some coordination between the local variables is necessary to make global layer activation work. In Figure 2 we see the major parts involved. Key set is the set of all values in a form suitable for serialization. `Environment` is the root of the generated class hierarchy. Objects of `ThreadContext` encapsulate all layers of one thread. As the name suggests, every thread has its own `ThreadContext`. Finally, the `Coordinator` takes care of coordination between `ThreadContext`. The boxes with the dotted pen width are typically only instantiated once per process.

Neither in `KeySet`, `ThreadContext` nor in the contextual values (=`Environment`) any locks or atomic values are needed. All coordination work is delegated to objects of the class `Coordinator`. Internally, both `ThreadContext` and `Coordinator` use the observer pattern. To completely decouple the coordination and the key set, callbacks are used.

Applications are not limited to a single object of the class `Coordinator` nor key set. CoElektra can be used within plugins or otherwise nested applications. Nevertheless, the following constraints apply: Every `Coordinator` is responsible for exactly one `KeySet`, every `ThreadContext` is bound to exactly one `Coordinator`, and again every contextual value is bound to exactly one `ThreadContext`.

C. Assignment

The assignment of contextual values adheres the expected semantics. Suppose `value` is specified to be an integer:

```
[/value]
type=long
```

Then, we can assign and use the contextual value as integer:

```
value = 8;
assert(value == 8);
value -= 2;
assert(value == 6);
```

The assignment has, however, additional overhead. For every change of the value, an underlying key set data structure is updated, too. This design decision is based on the assumption that customization happens rather seldom. This behavior has two desired properties. First, newly created contextual values always have an up-to-date value:

```
value = 5;
```

```
ThreadContext tc(c);
Value value(ks, tc);
assert(value == 5);
```

Second, the data structure key set is always up to date and its serialization leads to desired results.

D. Synchronization Points

Read-only use of contextual values is unsynchronized. To see layer (de)activations and assignments of other threads, the programmer has to define checkpoints. In the checkpoints global locks ensure sequential execution. Priority concerns are left to threading facilities of the operating system.

The explicit definition of checkpoints has an important advantage if an algorithm cannot deal with variables changed at any time: The programmer defines where updates can be done safely. For example, let us consider an algorithm that should terminate faster when the battery is low and thus enables the device to consume less power. First, we introduce a contextual value that defines the needed accuracy for the algorithm:

```
[/algorithm/%battery_low%/accuracy]
type=long
```

Some algorithms would allow us to change the accuracy at any time. Then we would not save power anymore, because of added synchronization overhead. Instead, we synchronize once and then calculate the task without any overhead:

```
void calculateTask(Algorithm::Accuracy const & accuracy)
{
    accuracy.context().syncLayers(); // sync accuracy
    for (long i=0; i<accuracy; ++i)
    {
        // calculate Task with given accuracy
        // values will not change within loop
    }
}
```

Note that `context()` returns the `ThreadContext` connected to `accuracy`. Using the methods (de)`activate` and `syncLayers` we define checkpoints. Every time they are passed, all contextual values changed in other threads are taken into account.

E. Thread-Based Layers

Layers that internally compare the current thread identification (ID) with selected thread IDs are a powerful tool. Even when such layers are activated globally, they influence only selected threads. For example, we can activate a layer only in a thread specified with the thread ID `selected`:

Implementing such layers is straight-forward: We have to check if the current thread ID equals the thread ID of a selected thread. The method `pthread_self()` returns the thread ID of the calling thread. If the current thread is the selected thread, we return an identification string. Otherwise we return an empty string to indicate that the layer is inactive.

```
class Thread : public kdb::Layer
{
public:
    string id() const { return "thread"; }
    string operator()() const {
        if (pthread_self() == selected) return "active";
        return "";
    };
private:
    pthread_t selected;
};
```

Such layers are also useful for changing the context for a pool of threads. We simply expand our idea from a single selected thread to a set of relevant threads. For example, if a pool of background threads all do the same calculations, we can influence all of them, but no other, with a single `activate()`.

F. Immediate Actions on Layer Activation

Sometimes we desire an immediate action when a layer is switched. For example, if we detected a tampering attempt of our hardware, we immediately want to remove all cryptographic private keys of the device:

```
void startupcode()
{
    Coordinator c;
    c.onLayerActivation<Tamper>([]()
        {removePrivateKeys();});
```

Then, the given function will immediately be executed once the layer `Tamper` is activated globally. It does not affect local `with` statements. Because the functionality is only for global activations, the method is part of the `Coordinator` interface. Such callback hooks are not only useful when we need to execute code as fast as possible on layer activation. They are helpful when activation/deactivation provides cleanup functionality.

For example, the shutdown of a web server can involve the joining of several threads. If a thread is blocked because of a system call, e.g., `read()` or `select()`, we have to send a signal in order to terminate the thread. Such specialties cannot be dealt with OOP-techniques (e.g. destructors) nor with exceptions-mechanisms (e.g. `finally`). They are, however, trivially solved using layer actions, e.g., a thread with a blocking `read()` or `select()` initializes itself in the following way:

```
pthread_t tid = pthread_self();
c.onLayerActivation<Shutdown>([tid]{
    pthread_kill(tid, SIGHUP);
});
// use of e.g. read() or select()
```

In the first line we get our own thread ID and store it in the variable `tid`. That is necessary, because we do not know which thread will activate the layer `Shutdown`. Then, we install an action on the layer activation of `Shutdown`. In it we kill our own thread to interrupt the blocking system call, e.g., `read` or `select`. Once the layer `Shutdown` is activated by any thread:

```
tc.activate<Shutdown>();
```

Not only contextual values, e.g. timeouts, are influenced, but also hooks are executed that are needed for the shutdown procedure. We can make sure that every thread reaches the exit point. The other available function `onLayerDeactivation()` can be used similar to `atexit`, but for layers, e.g., for cleaning up resources allocated on layer activation.

G. Implementation and Use

Finally, we want to describe the key aspects of our implementation design and how to take advantage of it. Our implementation is heavily based on C++ policy-based class design. Some behavioral aspects can be changed by policies. Because of the way we used policies in template arguments, they do not cause overhead compared to separately implemented classes.

For example, Elektra, which did not support multi-threading [7], is still available by using another policy class. Yet another policy class completely disables the context oriented features. CoElektra differs in using the new policy class `ThreadContext`. As a matter of fact, the contextual value classes are a C++11 **`using`** with correct policy classes applied:

```
template<typename T,
    typename PolicySetter1 = DefaultPolicyArgs
    /* + N more PolicySetters */ >
using ContextualValue = Value <T,
    ContextPolicyIs<ThreadContext>,
    PolicySetter1
    /* + N-1 more PolicySetters */ >;
```

Let us look at a different specification:

```
[/version]
type=boolean
readonly
opt=V
```

In this case, the class `Version` is specified as type `boolean`. Because of the property **`readonly`**, write attempts to the contextual value lead to a compilation error. The property **`opt`** specifies that the value is connected with a command-line parameter `-V`. For example, a `main` function that uses CoElektra:

```
int main(int argc, char**argv)
{
    KeySet ks;
    ksGetOpt(argc, argv, ks);
    KDB kdb;
    kdb.get(ks, "/");
    Coordinator c;
    ThreadContext tc(c);
    Environment<WritePolicyIs<ReadOnlyPolicy>>env(ks,tc);
}
```

The implementation of the method `ksGetOpt()` and the class hierarchy `Environment` are synthesized from the code generator. `ksGetOpt()` parses all arguments as specified with **`opt`**. The key database `KDB` reads the other parts from the program execution environment using `kdb.get()`. These parts are typically configuration files.

The `WritePolicyIs` of the example above changes the policy class for all contextual values for this instantiation of the `Environment`. Values in this hierarchy cannot be changed anymore. Instead, every attempt will lead to a compilation error. The class `Environment` forwards all policies, introduced in a **`using`** or as shown in `main`, to all generated contextual classes in the nested class hierarchy. When the developer wants to influence individual contextual values in every instantiation, the specification must be used instead.

IV. Evaluation

We benchmarked an application that uses CoElektra on a Raspberry Pi® Model B and on a hp® EliteBook 8570w using the central processor unit Intel® Core™ i7-3740QM @ 2.70GHz. The operating system was Debian GNU/Linux Wheezy 7.8 with the respective architecture armhf (Raspbian) and amd64. We used the compiler gcc 4.7.2-5 (+rpi1 on Raspbian). The systems were not altered for performance improvements, e.g. maximal number of file descriptors were left unchanged to their default 1024.

A. Case Study: HTTP Server

We developed an HTTP Server using the high-performance C++ web development framework CppCMS [8]. The target platform was a Raspberry Pi® Model B because of its low prize and power consumption. In this case study, we will show some functionality that is directly and elegantly expressible with CoElektra. Additionally, we will show that the performance (expressed as page replies per second) are hardly influenced when applications use CoElektra. From only 83 lines of specification 3500 lines of policy-based nested C++ classes and command-line option parsing code were generated with some affect on the size of the binary. The used caching technique, however, leads to much better execution times [7].

CoElektra is well suited to represent an HTTP session as contextual information. As first step, we need a specification:

```
[/sw/pi/%session%/language]
type=string
[/sw/pi/%language%/hello]
type=string
```

After we defined the layers that represents the language and a session, we are ready to open a session within the HTTP request handler (out is a stream to write the HTTP response):

```
tc.with<Session>(sessionid)([this](){
    out << "<html>\n"
           "<body>\n";
    out << "<p>Language: " << language << "</p>";
    tc.with<Language>(language)([this](){
        out << "<p>" << hello << "</p>";
    });
    out << "</body>\n"
           "</html>\n";
});
```

We create a thread-local context using `with<Session>`. Then, all contextual values are changed according the session, e.g. the contextual value `language`. Then, we can activate other layers, e.g. `Language`. When we output the contextual variable `hello`, the output depends on the user's language configuration.

Obviously, we do not want to lose session information of a user, e.g. the selected language. CoElektra easily fulfills such persistency requirements with the following code:

```
std::unique_lock<std::mutex> l = c.requireLock();
kdb.set(ks, "/");
```

In the code, we require a lock from the `Coordinator` and use Elektra's functionality to serialize all parts of the data structure `ks` using `kdb.set()`. The key set `ks` contains all contextual values, but the second argument allows us to restrict which parts are serialized.

Additionally, we were able to represent global changes of contexts. We used this feature for tamper detection. In our case study we used a passive infrared sensor HC-SR501 connected to a general-purpose input/output (GPIO). On tampering events, the information is displayed on the delivered web pages. To implement it we first specify a contextual value:

```
/sw/pi/tamper/%tamper%
```

One thread uses the system call `select` to wait for tampering events. Once a tampering event occurs, the next statement activates the layer `Tamper`. We already know that this action eventually changes all contextual variables in all threads. In our case study we used the contextual value `tamper` to inform users (Note that `tamper` is abbreviated to `t` and `context` to `c`):

```
select(fd+1, 0, 0, &fds, 0);
t.c().activate<Tamper>();
```

```
t.c().syncLayers();
if (t) out<< "tamper!!!";
```

Finally, we were able to arbitrarily multiplex GPIO using layer activations. Such requirements can be elegantly fulfilled using hardware profiles. A profile is a layer that does not distinguish between states, but between different setups. In our case study we had a configuration file with the following content:

```
/hw/pi/pi/gpio/folder = /sys/class/gpio/
/hw/pi/pi/gpio/tamper = gpio7
/hw/pi/elitebook/gpio/folder = ~/context/pi
/hw/pi/elitebook/gpio/tamper = tamper.txt
```

Given a hardware profile identification, e.g., read from an ordinary file or EEPROM, we can activate the correct hardware profile. On the laptop EliteBook, where no GPIO is available, we wrote test values into ordinary files that behave in the same way as the kernel interface. Apart from easier development, these hardware profiles allowed us to have different hardware setups with the same application running on it.

Additionally, for testing purposes, we took advantage of our approach and activated layers within unit tests, which works without target hardware: We get hardware abstraction for free.

B. *HTTP Replies*

First, we did benchmarks on the EliteBook. To measure the replies per seconds we used `httperf` with the arguments:

```
--hog --num-conn=600000 --rate=6000 --server localhost
```

We determined the rate by searching for the highest throughput without errors. Instead of the tampering by physical movement, we used a loop that tampers every N time units:

```
while (!shutdown)
{
    tc.activate<Tamper>();
    std::this_thread::sleep_for(N);
    tc.deactivate<Tamper>();
    std::this_thread::sleep_for(N);
}
```

We simulated a very high number of layer activation/deactivations. Even with only a nanosecond delay (N), no decay of replies per seconds was measurable. Only when the delay was completely removed, a considerable slowdown was experienced. We expect that this effect is the result of the nearly-always hold locks that cause starvation.

To explore the effect of long hold locks more closely, we tried to use the serialization of the key set. The serialization involves file access and naturally takes much longer. When accessing the key set, a lock causes all requests to wait. Even though expected otherwise, this way we got no decay of performance. Out of options for a realistic setup, we require a lock from `Coordinator` for a fixed time of ten milliseconds. Then, we vary the time L (during which no lock is held):

```
while (!shutdown)
{
    std::this_thread::sleep_for(milliseconds(L));
    t.syncLayers();
    std::unique_lock<std::mutex> l = c.requireLock();
    std::this_thread::sleep_for(milliseconds(10));
}
```

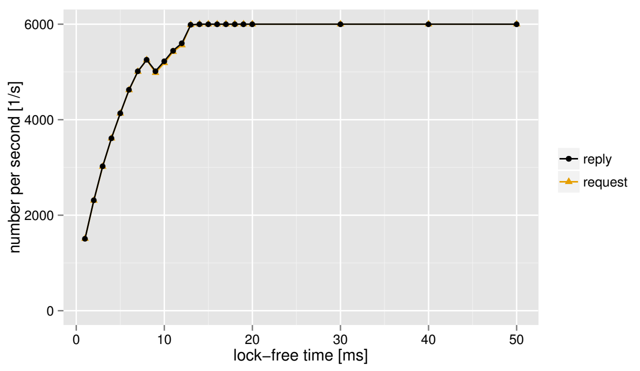

Fig. 3. HTTP requests and replies per second with increasing lock-free time (L) in ms. The time of 10 milliseconds corresponds to 50% lock-free time.

As we see in Figure 3 both the requests and replies rise with longer lock-free periods. With 14 milliseconds, i.e. 58.3%, unlocked time, we already get the full throughput time of 6000 replies per second. We conclude that CoElektra is sensitive to locks that are not released quickly. We expect this behavior to be a non-issue, because it is easy to avoid holding the lock of the `Coordinator` for such a long time.

On a single-threaded system, however, the picture looks entirely different: In the second benchmark the web server was running on the Raspberry Pi and httperf was running on the EliteBook with following arguments:

```
--hog --num-conn=15000 --rate=150 --server pi
```

The two computers were connected using a 100MB/s switch. The maximum throughput rate on this hardware is only about 150 replies per second, because of threefold reasons: The processor is much slower, the processor has only a single core, and the network stack adds additional overhead. Running `httperf` on the same hardware is not an option, because then the low resources would be reduced again.

As shown in Figure 4, in this setup a decay of performance is minimal, but clearly visible, when `activate()` and

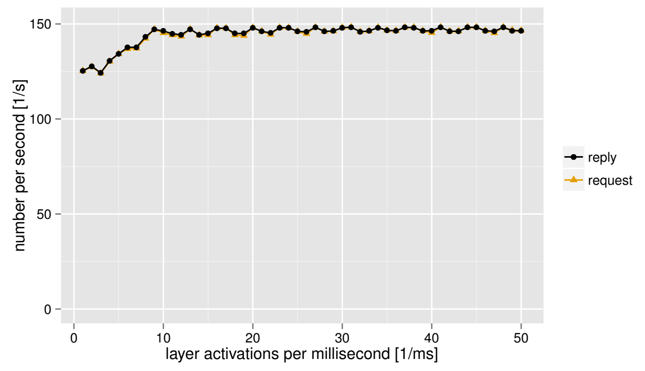

Fig. 4. HTTP requests and replies per seconds on Raspberry Pi with decreasing layer activations per milliseconds (N).

`deactivate()` is executed in a loop with a sleep time (N) of 13 milliseconds[1]. Such an effect is not surprising. The reason is that on the single-core processor, the background activity (layer switching) influences the main activity, because of the HTTP request handlers have to share the same core.

C. Performance Comparison

For the next benchmarks we used time-measurement with `gettimeofday`. We executed each benchmark eleven times for the box-plot. We used 100,000 invocations of different methods in CoElektra. Because the results on the Raspberry Pi are nearly identical, except of a large constant factor, the results are not shown here.

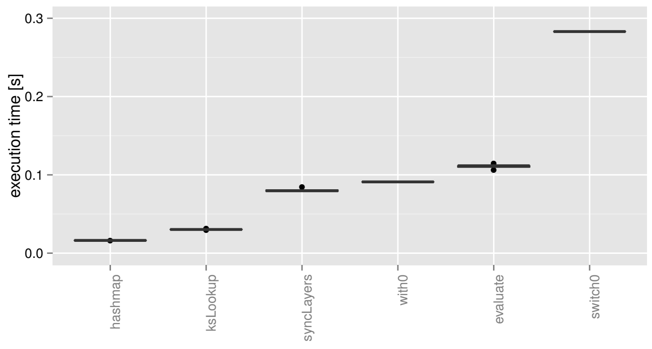

Fig. 5. Comparison of the most important operations on the EliteBook with linear scale in a box plot. Black dots are outside 1.5∗interquartile range.

As we see in in Figure 5 the C++11 hash map lookup (0.016 seconds) is twice as fast as Elektra's method `ksLookup` (0.03 seconds), which used to lookup values in the key set. This is not surprising as `ksLookup` has additional features such as cascading lookups and namespaces [9]. Very welcome is the property that `syncLayer` is fast (0.08 seconds).

In CoElektra the developer specifies which layers influence which contextual value. As we see in Figure 6 the number of influenced values play a crucial role: A `with` block (`with0`: 0.09 seconds), and the methods `activate` and `deactivate` (`switch0`: 0.28 seconds), that do not influence a single variable, execute in a short time (also shown in Figure 5). Note that for `withN` and `switchN` benchmarks only 50,000 loop iterations are necessary to perform 100,000 invocations. With a higher number of connected contextual vales, the costs increase linearly.

The method `evaluate` needs some explanation: It is used to replace all placeholders with the correct values of the layers. This method is a very common operation. In the benchmark the specification is 43 characters long and contains three placeholders. We see that this part central to CoElektra does not contribute significantly to the `with` and `activate` overhead if values are connected to it. Surprisingly, the propagation of all events to other `ThreadContext` is not expensive: `activate` and `deactivate` (=switch) perform nearly as good as `with`.

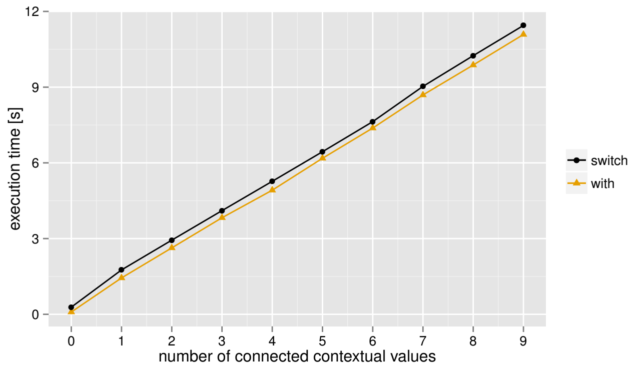

Fig. 6. Comparison of layer switches using the methods activate, deactivate and with. The number of connected contextual values is varied.

D. Resource Utilization

The stripped library `libelektra.so.0.8.10` only has a size of 109,912 bytes on amd64 and 98,456 bytes on armhf. The plugin for parsing INI files, needs another 22,760 bytes extra (libelektra-ini). In order to have a multi-process and thread-safe storage another 47,560 bytes are needed (libelektra-resolver). The library libxml2.so.2.8.0 that is often used for similar purposes (i.e., configuration file parsing and validation) has a size of 1,436,984 on amd64 and 1,196,108 bytes on armhf.

Elektra needs to allocate memory on the heap. To measure the amount we used the maximum resident set size. First, we started a program creating an empty key set on the EliteBook which needed 6660 kilobytes. For the other numbers given here, we removed that offset. With only two keys in a set, 84 kilobytes were needed. For ten thousand keys 1900 kilobytes were required, which should be sufficient for most use cases. For forty thousand keys 7612 kilobytes were needed.

On Raspberry Pi we get similar, but smaller, numbers. The program creating an empty key set needs 1568 kilobytes. With only two keys in a set, 60 kilobytes extra are needed. For ten thousand keys 1248 kilobytes were required and, finally, for forty thousand keys 4864 kilobytes were needed. We think that the smaller additional memory is due to the 32bit architecture: The data structure uses pointers extensively.

[1]It is pure coincidence that this number is similar to the one of the previous experiment.

V. Related Work

Jung et al. [10] applied code generation for embedded systems. Different from CoElektra they used partial evaluation. They assumed that the configuration is static, i.e. neither customizable nor context dependent. Additionally, they used libxml2 which needed to be removed by the partial evaluation, because of the much larger size as discussed in Section IV.

Watanabe and Takeno [11] describe an actor-based model for cross-context messages. Their approach enables them to asynchronously change the context with all messages received in the correct context. This work is of interest, when CoElektra is applied to actors instead of threads.

Costanza et al. [4] briefly mention `ensure-active-layer` that allows them to globally activate layers. Unfortunately, no explanation of the semantics is given. Their approach dynamically creates classes and inherits from layers, while in CoElektra layers are ordinary hand-written classes.

Riva et al. [12] unearthed that a hybrid mediator-observer is used in almost all of their surveyed COP systems. CoElektra is no exception and uses this pattern for thread synchronization.

In our previous work Elektra [7] we demonstrated that contextual values can have zero overhead on access compared to native C++ variables. CoElektra still has this property.

Löwis et al. [13] also claim that most context-dependency is contextual state. They introduce dynamic variables that are similar to our contextual values without serialization. Their proposal for implicit layer activations is comparable to our approach, but very costly: They add checks on every use.

VI. Conclusion

In this paper, we introduced semantics to efficiently and elegantly support context that is inherently global, e.g., changes of the physical environment. They fulfill a whole range of use cases in embedded and ubiquitous computing. So called thread-based layers retain previous semantics by limiting global activation to selected thread(s) in an efficient way.

We saw that our approach introduces a better multi-core processor support for context-aware ubiquitous computing. Our approach enables algorithms to concurrently read contextual values without any performance decay. Only actually switching context and synchronization points cause some overhead, but is efficient if the developer specified that only a reasonable number of contextual values are influenced.

In benchmarks we showed that continuous context switches do not have significant impact on a web server application. On a single-core processor with a high number of context switches the decay was noticeable. On a multi-core processor only an unrealistic long global lock, with its lock less than 58.3% released, reduced the number of replies per seconds.

Our contributions are:

1) CoElektra enables programmers to use contextual values in multi-threaded, embedded applications.
2) Our implementation is free software and can be downloaded from http://www.libelektra.org.
3) In a case study we described our experience with embedded hardware running a web server.
4) We analyzed the performance in multi-core and single-core setups and evaluated the memory footprint.

These contributions are significant, because up to now context-awareness had a much larger performance impact and were mainly available in non-embedded systems.

Acknowledgments

Many thanks to my colleague of my institute. I want to give distinct gratitude to Franz Puntigam for guiding me through the work, for being extraordinarily responsive to all my concerns and questions and for proof reading my papers. Additionally, I would like to thank the anonymous reviewers, Gergö Barany and Andreas Morhammer for a detailed review of this paper.

References

[1] H. Schippers, T. Molderez, and D. Janssens, "A graph-based operational semantics for context-oriented programming," in *Proceedings of the 2Nd International Workshop on Context-Oriented Programming*, ser. COP '10. NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1930021.1930027

[2] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-oriented programming: A software engineering perspective," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1801 – 1817, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412121200074X

[3] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, "A comparison of context-oriented programming languages," in *International Workshop on Context-Oriented Programming*, ser. COP '09. NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1562112.1562118

[4] P. Costanza, R. Hirschfeld, and W. De Meuter, "Efficient layer activation for switching context-dependent behavior," in *Modular Programming Languages*, ser. Lecture Notes in Computer Science, D. Lightfoot and C. Szyperski, Eds. Springer, 2006, vol. 4228, pp. 84–103. [Online]. Available: http://dx.doi.org/10.1007/11860990_7

[5] C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini, "Efficient control flow quantification," in *ACM SIGPLAN Notices*, vol. 41. ACM, 2006, pp. 125–138.

[6] E. Tanter, "Contextual values," in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS '08. NY, USA: ACM, 2008, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/1408681.1408684

[7] M. Raab and F. Puntigam, "Program execution environments as contextual values," in *Proceedings of 6th International Workshop on Context-Oriented Programming*. NY, USA: ACM, 2014, pp. 8:1–8:6. [Online]. Available: http://doi.acm.org/10.1145/2637066.2637074

[8] A. Beilis, http://cppcms.com, accessed January 2015.

[9] M. Raab, "A modular approach to configuration storage," *Master's thesis, Vienna University of Technology*, 2010.

[10] M. Jung, R. Laue, and S. A. Huss, "A case study on partial evaluation in embedded software design," in *Software Technologies for Future Embedded and Ubiquitous Systems, 2005. SEUS 2005. Third IEEE Workshop on*, May 2005, pp. 16–21.

[11] T. Watanabe and S. Takeno, "A reflective approach to actor-based concurrent context-oriented systems," in *Proceedings of 6th International Workshop on Context-Oriented Programming*, ser. COP'14. New York, NY, USA: ACM, 2014, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2637066.2637069

[12] O. Riva, C. di Flora, S. Russo, and K. Raatikainen, "Unearthing design patterns to support context-awareness," in *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, March 2006, pp. 5 pp.–387.

[13] M. von Löwis, M. Denker, and O. Nierstrasz, "Context-oriented programming: Beyond layers," in *Proceedings of the 2007 International Conference on Dynamic Languages*, ser. ICDL '07. NY, USA: ACM, 2007, pp. 143–156. [Online]. Available: http://doi.acm.org/10.1145/1352678.1352688