ELUS: a Dynamic Software Reconfiguration Infrastructure for Embedded Systems

Giovani Gracioli and Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
P.O.Box 476, 88040900 – Florianópolis – Brazil
{giovani,guto}@lisha.ufsc.br

***Abstract*—Dynamic software reconfiguration is the process of updating the system software during its execution. A dynamic software reconfiguration mechanism for an embedded system must be simple, transparent to applications, and use the minimum amount of resources (e.g. memory, processing) possible, since it shares resources with the target embedded system. We present EPOS LIVE UPDATE SYSTEM (ELUS), an operating system infrastructure for resource-constrained embedded systems. Through the use of sophisticated C++ static metaprogramming techniques, unlike the previous software reconfiguration infrastructures, ELUS provides a low-overhead, simple, configurable, and fully transparent software reconfiguration mechanism. Our experimental evaluation shows that the ELUS memory consumption, overhead, and reconfiguration time present better performance when compared to related work.**

I. INTRODUCTION

Dynamic software reconfiguration is a desirable feature present in the majority of current systems, from conventional environment computing (e.g. personal computers) to embedded systems. This characteristic allows the system software to be replaced at execution time, in order to correct bugs, add and/or remove features, and adapt the system to varying execution environments.

Wireless Sensor Networks, for example, can comprise a large number of small sensors scattered throughout a given environment, with few kilobytes of memory and low processing power [**?**]. Very often, collecting such sensors for in-lab reprogramming is unpractical. In this case, a dynamic software reconfiguration mechanism would be of great value, because it could reprogram the entire network remotely. Moreover, the mechanism should also be adaptable to the great variability of embedded system platforms in terms of memory capabilities, interconnecting (e.g. RS-485, CAN, ZigBee), and different processing power.

Nonetheless, a software reconfiguration infrastructure for a resource-constrained embedded system will be sharing resources with the target embedded application and must not disrupt the embedded system operation [**?**]. Dynamic software reconfiguration mechanisms for embedded systems can be split in three categories: (i) those based on binary code updating [**?**], [**?**], [**?**]; (ii) virtual machines [**?**], [**?**], [**?**]; and (iii) operating systems [**?**], [**?**], [**?**], [**?**]. In the first one, a bootloader or linker is responsible for receiving a new system image and updating the code. In the second one, software reconfiguration is done by updating the application script that is interpreted by the virtual machine. Finally, operating systems are designed to abstract a software update. Usually, OSs are organized as reconfigurable modules and create an indirection level between the application and module through tables and/or pointers. Updating the software in this case is done by changing the module address inside these tables/pointers.

In this paper we present EPOS LIVE UPDATE SYSTEM (ELUS), a dynamic software reconfiguration infrastructure for resource-constrained embedded systems. ELUS is built within EPOS component framework, around the *remote invocation* aspect program [**?**]. The main features that make ELUS different from previous OS infrastructures are:

- **Configurability:** system components[1] can be marked as reconfigurable or not at compile-time. For those components that are not marked as reconfigurable, no overhead, in terms of memory and processing, is added in the final system image.
- **Small Memory Consumption:** a reconfigurable component has an additional memory overhead around 1.6Kb of code and 26 bytes of data per component, which is a lower memory consumption compared to previous work.
- **Transparency and Simplicity:** unlike other OS infrastructures, a system component does not need to register and unregister its methods during its initialization or destruction. The reconfiguration is carried out by a transport protocol, named ELUS TRANSPORT PROTOCOL (ETP). Moreover, the reconfiguration infrastructure and the software updating are fully transparent to applications.
- **Message Structure:** by using the EPOS framework message structure, passing arguments and return values among components' methods are done in an efficient way, about 5 times faster than previous work.
- **Reconfiguration:** ELUS allows the developer to update both application and operating system. In some OSs, like Contiki [**?**] for example, only applications or part of the OS are updatable. The time spent in a reconfiguration also represents good performance.

The remainder of this paper is organized as follows. Section II presents EPOS metaprogrammed framework. In Sec-

[1]EPOS components are encapsulated in C++ classes with well-defined interface and behavior.

tion III the design and implementation of ELUS is presented. Section IV shows the experimental evaluation. Section V makes a result analysis by comparing ELUS with related work. Finally, Section VI concludes the paper.

II. EPOS FRAMEWORK

EMBEDDED PARALLEL OPERATING SYSTEM (EPOS) is a multi-platform, component-based operating system, in which traditional operating system services are implemented through adaptable, platform-independent *System Components* [**?**]. Platform-specific support is implemented through *Hardware Mediators* [**?**]. Mediators are functionally equivalent to device drivers in UNIX platforms, but do not compose a traditional hardware abstraction layer. Instead, they make use of platform-independent interfaces to sustain their interface contract between components and hardware. Through the use of static metaprogramming and function inlining, mediator code is *dissolved* into components at compile-time.

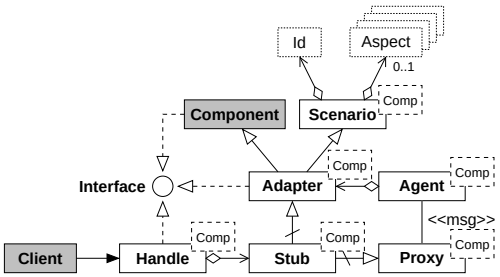
EPOS has a highly scalable architecture that is molded to accomplish the needs of applications. It supports aspects through a scenario adapter mechanism [**?**]. Distinct combinations of system components and scenario aspects lead to different software architectures. In this context, EPOS implements a framework which defines how components can be arranged together into a functioning system. EPOS framework is realized by a C++ static metaprogram, which is executed in the course of a system instance compilation, adapting selected components to coexist with each other and with applications.

An overview of the EPOS metaprogrammed framework is depicted in Figure 1(a). A Client (an application or component) wants to invoke a method of a component (e.g. Thread, UART, etc). The parametrized class HANDLE receives a system component as parameter. It verifies if the object was correctly created and forwards method invocations to STUB element.

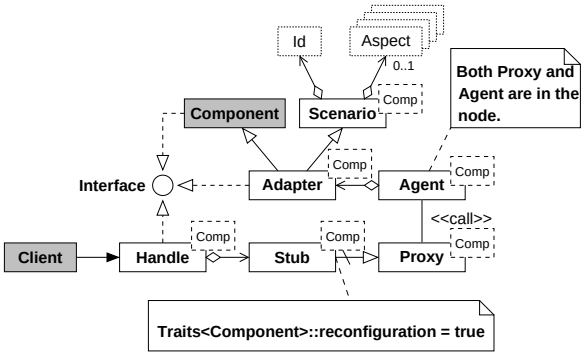
The STUB element is a parametrized class responsible for verifying whether the *aspect of remote invocation* is active for that component (the remote invocation aspect is selected by the component through its *Trait Class* [**?**]). If it is not, the STUB will inherit the component's scenario adapter. Otherwise, a stub's specialization, namely *Stub<Component, true>*, will inherit the component's PROXY. Therefore, when *Traits<Component>::remote = false*, this makes HANDLE take the scenario adapter as STUB, while making it true makes HANDLE take PROXY.

The PROXY is responsible for sending a message with the component method invocation to its AGENT. A message contains the object, method, and class IDs that are used by AGENT to invoke the correct method, associating them to a method table. The object ID is used to get the correct object before calling the method. The AGENT receives the message and invokes the method through ADAPTER.

ADAPTER class is responsible for applying the aspects supplied by SCENARIO before and after the real method call. Each instance of SCENARIO consults the component's *Traits* to verify which aspects are enabled for that component,



(a) EPOS metaprogrammed framework.



(b) Framework for software reconfiguration.

Fig. 1. (a) EPOS metaprogrammed framework overview [**?**]. (b) EPOS metaprogrammed framework for software reconfiguration.

```
1 namespace Application {
2     // Export the system component to application namespace
3     // as a Handle parameter
4     typedef Handle<System::Component> Component;
5 }
```

Fig. 2. Exporting a system component to application namespace.

aggregating the corresponding scenario aspect. When an aspect is not selected for the component, an empty implementation is used. In this case, no code is generated in the final system image, as an aspect weaver in traditional Aspect-Oriented Programming (AOP).

The framework structure is totally transparent to applications due to the use of two namespaces: one for applications and another one for the system. At compilation-time, system components are exported to the application's namespace as demonstrated in Figure 2. Consequently, a method invocation from application to a system component is done in the same way (*component->method()*), but instead of calling the real method, the call passes through the framework. System components which want to invoke methods from reconfigurable components, must use the application namespace as well (*Application::Component*).

We have observed in this work that the PROXY/AGENT structure found in the framework creates an indirection level between method invocation from client to system components with the remote aspect enabled, isolating the components and making them memory position independent. In this scenario, from the client perspective, a remote method invocation is carried out without knowing the component location. This

isolation could also be applied for a dynamic software reconfiguration, since only the AGENT framework member is aware of the component memory location. Unlike the remote method invocation aspect, we propose replacing the method invocation over the network between PROXY and AGENT to a simple method call between them. Consequently, the same indirection level could reside at same address space of a node. Figure 1(b) demonstrates this scenario in the framework structure.

III. EPOS LIVE UPDATE SYSTEM (ELUS)

In order to perform a safe and correct software reconfiguration without compromising the system, there are three **main** requirements that must be satisfied [**?**]:

1) **Quiescent State:** before executing a software reconfiguration, the system must reach a consistent state, which is named quiescent state. In a multithread environment such as EPOS, a quiescent state of a component is reached when none of the Threads are calling methods from it.
2) **State Transfer:** after reaching a quiescent state, the state of the old component (attributes) must be transferred to the new one, when there are attributes. The state transfer can be performed through memory copy from old to new component, by creating a new object and passing attributes' values from the old component to the new object's constructor, or by accessing the *set* and *get* methods from the components.
3) **Reference Redirection:** in order to keep the system consistent, references pointing to old component code must be redirected to new component code.

In addition to these requirements, an embedded system has specific restrictions. Therefore, a software reconfiguration mechanism competes for resources with the target embedded system. Thus it should use the minimum amount of resources possible. Consequently, low memory, processing, and power consumptions are **desirable** requirements

A. Assumptions

Based on the software reconfiguration requirements, ELUS defines the following assumptions:

- The reconfiguration unit is a component. A component is marked as reconfigurable or not at compile-time, and hence only those marked as reconfigurable can be updated. For those that are not marked as reconfigurable, no overhead (in terms of memory and processing) is added in the final system image. Yet, all system can be marked reconfigurable.
- The quiescent state of a component must be reached before executing a reconfiguration. In this way, a reconfiguration will be done consistently and will not disrupt the system behavior.
- The system hardware does not change. Both old and new versions of a component execute in the same hardware platform.

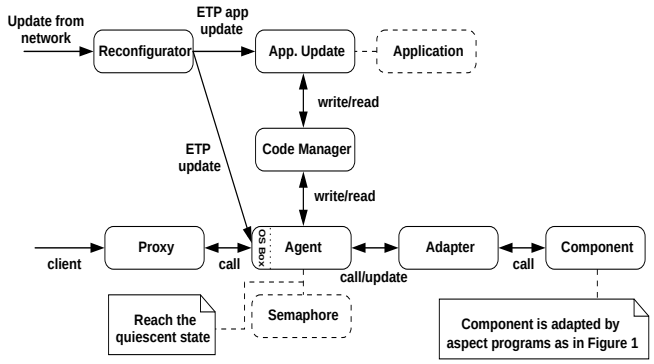


Fig. 3. ELUS architecture overview.

- It is up to the developer to ensure that the new component code does not remove any method being used by other components during a reconfiguration.
- Reconfigurable methods of a component must be declared virtual. When a method is declared virtual, the C++ compiler generates a virtual method table (*vtable*) for objects of that component. This table contains the component methods' addresses. A vtable supports runtime binding, and will be used to redirect the references from old component code to new component code.
- The access of a component attribute must exclusively be done by invoking *set* and *get* methods. Thus, the state transfer uses the *get* method to pass attribute values from the old component to the constructor of the new component object. After that, the old component object is deleted.

B. Architecture Overview

An overview of the ELUS architecture is presented in Figure 3. The original framework infrastructure was extended to support software reconfiguration. The invocation of a component method of the client (an application or a component) passes through PROXY which sends a message to the AGENT. After the method execution, a message with the return value is sent back to the client. An indirection level is created between the client and the real component, making the AGENT the only one aware of the component's position in the system memory. The OS BOX at AGENT was added to control the access to the component method through a synchronizer (*Semaphore*) for each component, avoiding the invocation of a component method while it is being reconfigured. It also acts as an entry point for invoking methods of a reconfigurable component, calling an AGENT method through a methods table. The AGENT forwards the invocation to the ADAPTER which calls the real component method through the object vtable.

The framework is a metaprogram which means that all dependencies between a component and execution scenarios are resolved during the system compilation. Nonetheless, the PROXY element is *dissolved* into the client by the compiler. In this way, calls from PROXY to AGENT are done directly from client to OS box. In the generated code, the PROXY, HANDLE,

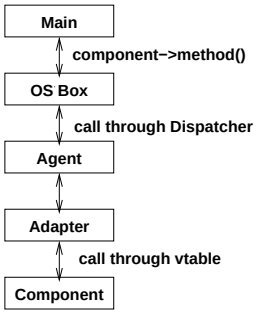


Fig. 4. A component method invocation through the Framework structure after compiling the system.

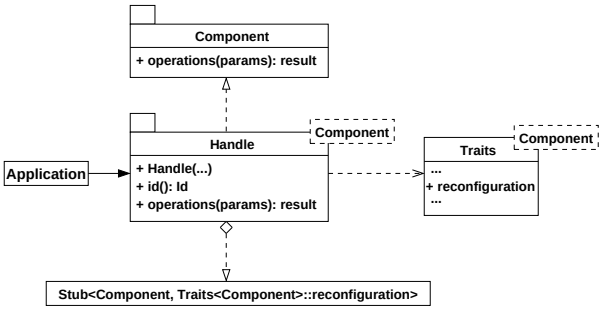


Fig. 5. ELUS Handle element.

and STUB elements do not exist.

Figure 4 exemplifies a method invocation using the ELUS framework. The application directly calls the OS Box from the main function. The OS Box accesses the AGENT method through a methods table (*Dispatcher)* and calls the correct method. The AGENT recovers the received parameters and calls the ADAPTER. Finally, ADAPTER calls the real component method using the object's vtable.

C. ELUS Framework Elements

The HANDLE ELUS framework element is depicted in Figure 5. It is responsible for verifying if a component was marked as reconfigurable or not by accessing the component Trait class. It has the same interface (operations) as the component, but method invocations are forwarded to STUB (*stub->method()*). The STUB element, however, will inherent the PROXY element if the reconfiguration is enabled. Otherwise, it will inherent an ADAPTER.

Figure 6 shows the PROXY and AGENT elements. These elements will only be present if the reconfiguration is enabled for a component. PROXY realizes the component methods. It uses the MESSAGE class in order to pass a method invocation to AGENT. When PROXY receives a request from HANDLE, it fills out a message by attaching the method and component IDs and parameters in a message.

The MESSAGE class offers an interface for attaching and detaching parameters and return values (methods *in* and *out*). These methods are also used in AGENT to recover parameters or to return a value. The invoke MESSAGE method calls the OS BOX function, which uses an array (*Dispatcher*) in order

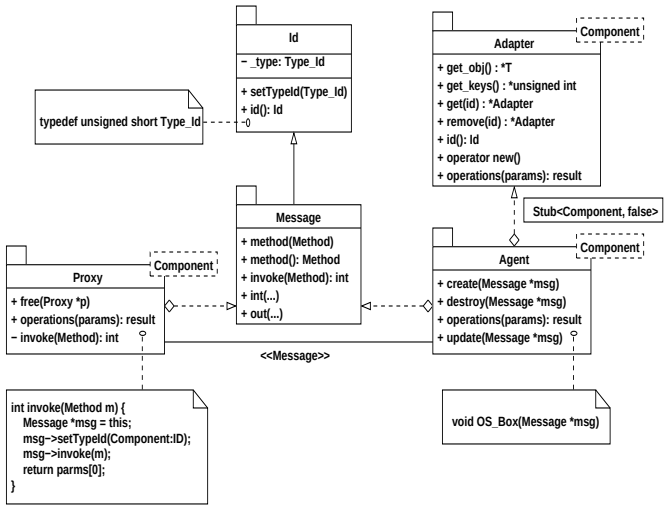


Fig. 6. ELUS Proxy and Agent elements.

```
1  typedef void (Dispatcher)(Message *);
2  Dispatcher* services[LAST_TYPE_ID + 1][MAX_METHODS] = {
3        { &Agent<Component 1>::create,
4          &Agent<Component 1>:::destroy,
5          &Agent<Component 1>::method1,
6          &Agent<Component 1>::method2,
7          //  ...
8          &Agent<Component 1>::update },
9        { &Agent<Component 2>::create,
10         &Agent<Component 2>::destroy,
11         &Agent<Component 2>::method1,
12         &Agent<Component 2>::method2,
13         //  ...
14         &Agent<Component 2>::update, }
15       //  ...
16 };
17 static Semaphore quiescent_state[LAST_TYPE_ID + 1];
18 void OS_Box(Message *msg) {
19       Dispatcher *d;
20       d = *services[msg->id().type()][msg->method()];
21        quiescent_state[msg->id().type()].p();
22       d(msg); // call a method
23        quiescent_state[msg->id().type()].v();
24 }
```

Fig. 7. OS Box and Dispatcher implementations.

to deliver the call to AGENT. Figure 7 shows the dispatcher and OS Box implementations.

The AGENT element is responsible for calling the ADAPTER methods and reconfiguring a component through the **update** method. A Thread, called RECONFIGURATOR, created during the system bootstrapping, receives a reconfiguration request and the new component code from the network (RS-232 or radio for example). This request is sent to the AGENT in the form of a MESSAGE using the ELUS TRANSPORT PROTOCOL (ETP) through the OS BOX as well.

D. Elus Transport Protocol (ETP)

ELUS TRANSPORT PROTOCOL (ETP) is a protocol for receiving reconfiguration messages. Figure 8 shows these messages. A control field identifies the reconfiguration type. There are six possible reconfigurations: (a) those for adding a method into a component; (b) those for removing a method

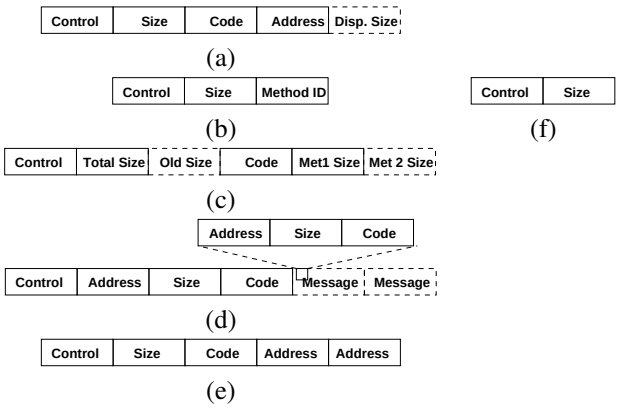


Fig. 8. ETP messages overview. (a) Method adding message (b) Method remove message (c) Component update message (d) Address update message (e) Application update message (f) Attribute adding message.

from a component; (c) those for updating a component; (d) those for updating a specific memory address; (e) those for updating the application; and (f) those for adding an attribute into a component.

AGENT uses the control field in order to identify which reconfiguration type is sent by the RECONFIGURATOR. The control field also identifies if the new component or method code is larger than the old one. If it is, AGENT allocates new memory for the new code. If it is not, the AGENT overwrites the old code. ETP is a simple protocol, allowing an easy integration with data dissemination protocols [**?**], [**?**], [**?**], for instance.

E. Reconfiguration Process

The reconfiguration process is done by the AGENT **update** method, which receives a MESSAGE in the ETP format. Figure 9 exemplifies a component update reconfiguration. The RECONFIGURATION sends an ETP message to OS BOX informing of a component update. The OS BOX reaches the quiescent state by calling a semaphore *p()* operation. After that, the AGENT reads the message fields (method *in*), and recovers the object(s) associated with the component ID and its vtable address (**addr = *reinterpret_cast<unsigned int>(&t)*). The control field stores information if the new component code is larger than the old one. If so, the AGENT allocates a memory space for the new code through a CODE MANAGER. Otherwise, the new code is written in the same position of the old one. In the end, AGENT updates the methods addresses in the vtable and releases the component semaphore by calling the *v()* operation.

CODE MANAGER is a class that manages the system code memory. It exports a simple interface to components which want to allocate, release, write, or read bytes to or from the code memory. CODE MANAGER is important because it abstracts memory dependencies among platforms (e.g. harward or von neuman architecture), meaning that each architecture should have its own CODE MANAGER.

In order to support an application update, we implemented a special component, called APPLICATION UPDATE. This

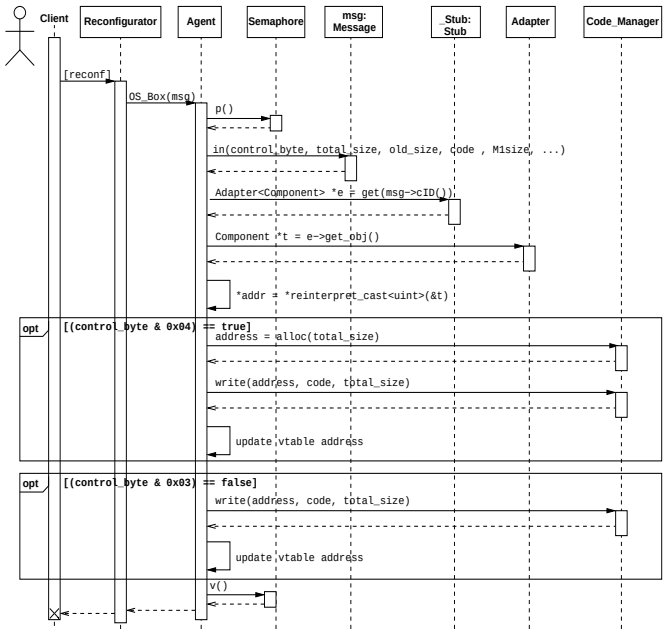


Fig. 9. Sequence diagram of the Agent update method when performing a component updating.

component is responsible for receiving an application update request in ETP format and updating the application code as needed. It can be reconfigurable as well. APPLICATION UPDATE uses two ETP messages: application update message and address update message. The control field identifies which updating type is being requested.

A new component code is compiled, linked with the generated system image, and its code sent over the network. The new component code is linked using the OS BOX address, which is the entry point for calling methods through the framework infrastructure. Therefore, the OS BOX is the only system element that **is not** reconfigurable.

IV. EXPERIMENTAL EVALUATION

A. Synthetic Benchmark Configuration

In order to corroborate design choices, we have evaluated ELUS in terms of three metrics: memory consumption, overhead associated with indirection level composed by the framework structure and virtual method table, and time spent in a reconfiguration. We have used GNU g++ compiler 4.0.2 and GNU objdump tool 2.16.1 to generate the system and analyze these metrics, respectively.

The platform used was the Mica2 mote [**?**], composed by an 8-bits AVR Atmega128 microcontroller, with 4KB of RAM memory, 128KB of flash memory, 4KB of EEPROM, and a set of peripherals including radio communication.

B. Memory Footprint

In the first evaluation, the reconfiguration support was enabled only in the *Chronometer* component that has 8 reconfigurable methods [**?**]. The objective was to measure the memory overhead added by the ELUS infrastructure. Table

I presents the memory consumption of the ELUS elements in this scenario. The framework has needed about 2.6kb of code memory, 43 bytes of data for *Dispatcher* and control attributes, and 52 bytes of non-initialized data for hash table in the SCENARIO framework element. RECONFIGURATION needs 70 bytes of data due to a buffer for receiving data from the network. This buffer was configured with size of 40 bytes. The APPLICATION UPDATE component uses 210 bytes of code memory. The total memory consumption for this case is about of 3.5Kb of code memory and 148 bytes of data.

TABLE I
ELUS MEMORY CONSUMPTION.

ELUS Elements	Section Size (bytes)			
	.text	.data	.bss	.bootloader
Reconfigurator	410	0	70	0
Code Manager	16	0	2	375
App. Update	210	0	0	0
OS Box	76	0	0	0
Framework	2620	43	52	0
Total	3452	43	124	375

The second memory benchmark evaluated the memory consumption of individual methods in the framework for a generic component. This test only considers the memory consumption of the framework, disregarding the memory needed by the component itself. The objective was to measure the memory overhead added for a component when it is marked as reconfigurable. Table II shows the evaluation.

TABLE II
MEMORY CONSUMPTION OF INDIVIDUAL METHODS IN ELUS FRAMEWORK.

Framework Method	Section Size (bytes)	
	.text	.data
Create	180	0
Destory	138	0
Method without parameter and return value	94	0
Method with one parameter and without return value	98	0
Method without parameter and with return value	112	0
Method with one parameter and return value	126	0
Update	1250	0
Dispatcher	0	2 X (methods)
Semaphore	0	18
Minimal Size	1662	26

There are three methods that are always added by the framework: *create* (constructor), *destroy* (destructor) and *update* (responsible for reconfiguring the component). Moreover, there is a table of AGENT methods (*Dispatcher*). Each method inside *Dispatcher* occupies 2 bytes. The other values present the overhead associated with a specific method of a component. For instance, if a component method passes one argument and receives a return value, there is memory consumption implicitly due to the MESSAGE structure for passing the arguments and return value. Also, each component must have a semaphore to control its access, which occupies 18 bytes in the data section. Thus, the minimal memory consumption (with create, destroy, update, and one method without parameters and return value) for a reconfigurable component is about 1.6Kb of code and 26 bytes of data. Equation 1 can be used for calculating the total framework memory consumption for a component.

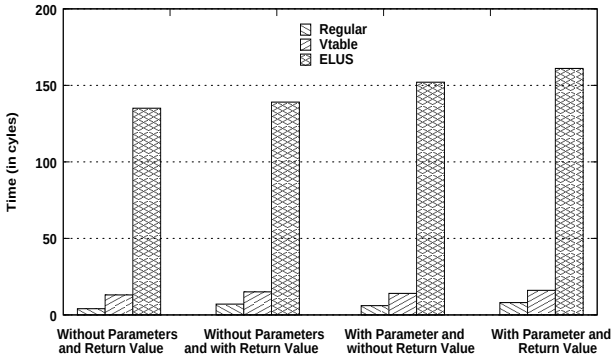


Fig. 10. Comparison of invocation time among a regular method invocation, through vtable and through ELUS.

$$Size_c = \sum_{i=1}^{n}(Method_i) + Create + Destroy + Update \quad (1)$$

C. Method Invocation Overhead

The indirection level from the application to a component method creates an overhead. Before calling the real method, AGENT should get the arguments from MESSAGE, recover the object from the hash table, and finally call the method using the object's vtable. Figure 10 presents a comparison among four types of methods: (i) without receiving parameters and returning value; (ii) without receiving parameters, but returning value; (iii) receiving one parameter, but without returning value; and (iv) receiving one parameter and returning value. For each type, there is a comparison among ELUS, using an object vtable, and through a regular method invocation. The method invocation overhead was about 10 times worse than using a virtual method table. As will be shown in Section V, this overhead is lower than in related work.

D. Reconfiguration Time

Table III presents the reconfiguration time in two component update scenarios, when the new component code is larger than old and when the new component code is less or equal to the old one. In this test the time spent in the RECONFIGURATOR for calling the OS BOX, the time spent in the OS BOX for accessing the *Dispatcher* and calling the AGENT update method, and the time spent by the AGENT update method on getting the arguments from message and executing a reconfiguration were considered. The time needed for receiving data from the network in the RECONFIGURATOR and the time for writing data to the flash memory were not considered because these times are also not used in the related work.

TABLE III
TIME (IN PROCESSOR CYCLES) FOR A COMPONENT RECONFIGURATION.

	Time (in cycles)	
	Update same position	**Update new position**
Agent Update	238	290
Reconfigurator	65	65
OS Box	59	59
Total	362	414

RECONFIGURATOR consumes 65 processor cycles to call the OS BOX. The OS BOX needs 59 cycles for calling the *p()* and *v()* semaphore operations, and AGENT update method. Finally, the update method spends 238 cycles performing an update in the same position as the old component code, and 290 cycles to update a component with new code larger than the old one.

V. RESULT ANALYSIS

This section discusses the ELUS results by comparing them to related work. Primarily, it will be presented the main operating systems that support software reconfiguration in embedded systems domain and after they will be compared to ELUS.

A. Related Work

TINYOS is considered the most popular OS for WSNs. It is an event-driven OS and originally does not support software reconfiguration [**?**]. However, several works have been done in order to support reconfiguration in TINYOS [**?**], [**?**], [**?**]. MANTISOS is another well-known OS for WSNs, but does not support software reconfiguration [**?**].

NANO-KERNEL allows dynamic reconfiguration of application and kernel components by decoupling the kernel data objects from the logical kernel algorithms [**?**]. The kernel address space is split up in two parts, the nano-kernel core and kernel devices, such as scheduler and memory manager. The nano-kernel core is responsible for creating an indirection level between application and kernel devices. In this way, when an application invokes a function from a kernel device or a kernel device invokes a function from another kernel device, the nano-kernel core redirects the call to the appropriate kernel device and returns the result to the caller. The nano-kernel core and the kernel devices communicate through a specific interface. In system boot, all kernel devices need to be loaded, and their interfaces need to be initialized with the appropriate methods. As the nano-kernel core receives all method calls from the interface, it must check if the kernel device has permission to access the data object, thus incurring an overhead to the system.

RETOS implements a mechanism for dynamic reconfiguration of system modules through a dynamic memory relocation and linking at run-time [**?**]. The relocation process extracts the information from global variables and functions at compile-time, creating meta-information (hardware-dependent information) about the module that is being updated. This information is placed in a RETOS file format that is used by the system kernel to replace every accessible address of a module when performing a module load. The OS also keeps a table that allows modules and applications to access other module functions. A module registers, unregisters, and accesses functions through this table. On the other hand, an application accesses the module function through a system call which invokes the required function by accessing this table.

CONTIKI is an OS for WSNs that implements special processes, called *services*, which provide functionality to other processes [**?**]. Services can be replaced at run-time through a *stub interface* that is responsible for redirecting function calls to a *service interface*, which holds the actual pointers to the functions implementing the corresponding services. The *service interface* also keeps track of versions during updates. Contiki is not full reconfigurable, that is, it only allows the updating of some system parts.

SOS is an operating system for sensor networks that allows nodes to be updated on-the-fly [**?**]. The OS is built around modules that can be inserted, removed, or replaced at run-time. By using relative calls, the code in each module becomes position independent. References to functions and data outside the scope of a module are implemented through an indirection table and, in some cases, are simply not allowed. ELUS is conceptually similar to the previously presented works, but the framework infrastructure takes advantage of static metaprogramming techniques to eliminate part of the overhead associated with indirection tables, thus yielding a slimmer and configurable mechanism, and also achieving application transparency. Table IV shows an overview of the reconfiguration process in the analyzed embedded operating systems.

TABLE IV
RECONFIGURATION PROCESS CHARACTERISTIC IN THE ANALYZED EMBEDDED OSS.

OS	**Reconfiguration Process**
TINYOS	No direct support
MANTISOS	There is no support
NANO-KERNEL	Reconfigurable modules
RETOS	Dynamic relocation and runtime linking
CONTIKI	Reconfigurable modules
SOS	Reconfigurable modules
EPOS/ELUS	**Selected reconfigurable components**

B. Discussion

The minimum ELUS memory consumption is about 1.6Kb of code and 26 bytes of data per system component. CONTIKI needs 6Kb of code and 230 bytes of data [**?**]. SOS occupies more than 4Kb of code for managing the modules and the module table occupies 230 bytes of data. ELUS presents good performance due to static metaprogramming, which solves all dependencies between the framework, components, and application at compile-time. Moreover, ELUS allows the developer to choose the reconfigurable components without imposing overhead in the final system instance. Therefore, the generated system only contains the code and data really necessary.

Regarding the method invocation overhead, SOS takes 21 cycles for calling a function of a reconfigurable module and

takes 12 cycles for calling a kernel function. However, the communication between modules (data exchange) is carried out by sending and receiving messages in a buffer, which takes 833 cycles [**?**]. CONTIKI has a worse performance because the stub should find the service interface before calling a function of a reconfigurable module. This is done by comparing a string of characters. After that, the function is accessed by a function pointer [**?**]. In a comparison, CONTIKI was 4 times slower than SOS [**?**]. RETOS and NANO-KERNEL also present an overhead associated with the indirection level. ELUS has the advantage of using the framework structure for passing arguments and return value, which is 7 times faster than SOS.

Finally, in a reconfiguration, SOS removes the old module and registers the new module and its event handler. It takes 621 cycles to perform these activities, disregarding the times for receiving the data from network and writing the code to flash memory [**?**]. CONTIKI also has a similar performance. It starts sending a message for removing the service interface, makes the state transfers between the old and new modules, and ends registering the new service interface. RETOS uses dynamic relocation and linking at run-time. There are meta-data that contain information about modules. These meta-data are transmitted together with the new code, which increase the data transmitted over the network and the reconfiguration time. ELUS does not need to register and unregister components at run-time. All dependencies are solved at compile-time. It takes about 500 cycles for performing a reconfiguration. ELUS also decreases the data sent to network by using the ELUS TRANSPORT PROTOCOL, which needs from 2 to 6 extra bytes.

VI. CONCLUSIONS

This paper presented ELUS, a low-overhead OS infrastructure for resource-constrained embedded systems. ELUS was developed around EPOS component framework, which was modified in order to enable confinement and isolation of system components in physical modules, thus making them memory position independent. The infrastructure also allows the developer to choose at compile-time which components can be reconfigurable. For those components marked as not reconfigurable, no overhead is added in the final system image. Moreover, ELUS is totally transparent for applications and by using the ELUS TRANSPORT PROTOCOL, the software reconfiguration becomes simple and easy to integrate with other protocols, such as a data dissemination protocol.

Results have shown that ELUS consumes less memory, adds a lower overhead, and has a better reconfiguration time in comparison to related work. The next step in the project is to integrate a data dissemination protocol with ELUS in order to support code distribution over the network.