A Design Patterns-Based Middleware for Multiprocessor Systems-on-Chip

Jean Carlo Hamerski*†, Geancarlo Abich‡, Ricardo Reis‡, Luciano Ost§, Alexandre Amory*
*Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, (PUCRS), Porto Alegre, Brazil
†Instituto Federal de Educação Ciência e Tecnologia do Rio Grande do Sul, (IFRS), Porto Alegre, Brazil
‡PGMICRO Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
§Wolfson School of Mechanical and Manufacturing Engineering, Loughborough University, UK
jean.hamerski@restinga.ifrs.edu.br, {gabich,reis}@inf.ufrgs.br, l.ost@lboro.ac.uk, alexandre.amory@pucrs.br

***Abstract*—Current multiprocessor systems might comprise dozens of processors, requiring a runtime management to provide performance while complying with system's constraints such as energy consumption, thermal balance, and fault tolerance. Self-adaptive multiprocessor systems have been proposed to cover some key aspects such as hardware abstraction, programming models and modular software architecture. This paper provides a modular middleware to assist the development of self-adaptive services and applications on MPSoC environments. Based on key design patterns, the proposed middleware uses highly efficient features of object-oriented programming for embedded systems and specific compiler optimization options. The results show a case study of a self-adaptive application on top of the proposed middleware. Additional experiments demonstrate a reduction in the execution time from 3% up to 19%, presenting a memory footprint overhead of 8.7% when compared to previous middleware without the features to ease modular software design.**

I. Introduction

The constant increase of applications' complexity along with technologies constraints (e.g., power and memory wall) influenced the evolution of the embedded systems from a single core to multiprocessor architectures. Multiprocessor system-on-chip (MPSoCs) provide parallel processing capabilities, aiming at covering the increasing requirements of emerging applications. Resulting complexity calls for a flexible and self-adaptive system, which must handle critical design constraints to address multiple complex applications competing for resources in the multiprocessor system.

Self-adaptive approaches have been proposed to cover the requirements of operating systems and applications at runtime [1–6]. Software modularity, object-oriented programming, and software design patterns are examples of techniques used to provide systems with self-adaptive property. The software modularity aims to separate the software design from the other system elements that are architecture dependents. Object-oriented programming provides known advantages such as code reuse and encapsulation. Software design patterns are widespread solutions to common problems in operating systems, which rely on portable code that can be reused in many different situations. Some works target to improve the software adaptability in embedded systems, focusing on software development [4] and security issues [6]. Although well-known in several segments, the applicability of these techniques is not well explored in the multiprocessor systems domain.

Middleware approaches leverage patterns and techniques to bridge the gap between the functional requirements of applications and the underlying architecture [7, 8]. This approach is used by solutions in the most diverse distributed environments, such as the DDS[1] (Data Distribution Service) for real-time systems environments and ROS[2] (Robot Operating System) for robotic environments. Although the client nodes are designed to memory-constrained environments, these solutions typically make use of more burdensome infrastructure for centralized management roles (e.g., broker), which are usually hosted on a node without memory restrictions.

The main contribution of this paper is to demonstrate the design of a middleware for proposing modular self-adaptive services in MPSoCs with small memory. The proposed middleware has 17.03 Kbytes of footprint. The modular features for self-adaptive services and applications are achieved by incorporating best practices of object-oriented programming, the publish-subscribe programming model for distributed systems [8], and design patterns which are suitable to MPSoCs with small memory. The proposed middleware is public available[3].

The rest of this paper is organized as follows. Section II presents related works. Section III provides recommended design patterns and best practices regarding to the middleware design. Section IV details the the proposed middleware. Section V presents the experimental setup and discusses the results. Section VI points out the conclusions and provides directions for future works.

II. Related Works

Several approaches are proposing the use of design patterns and modular techniques to address the design of self-adaptive systems in different computing systems.

Ramirez and Cheng [1] conduct a study comprising project implementations to harvest adaptation-oriented design patterns that support the development of adaptive systems in general domains. A subset of collected design patterns is evaluated in an adaptive news web-server case study. In this context, Abuseta and Swesi [2] propose a set of design patterns for modeling and designing self-adaptive software systems based on IBM MAPE-K multiple control loop issues. To evaluate

[1]http://www.omg.org/spec/DDS/
[2]http://www.ros.org
[3]https://bitbucket.org/jeanhamerski/mqsoc/

978-1-5386-7431-4/18/$31.00

TABLE I. RELATED WORKS COMPARISON

Reference	Domain	Approach	Case Study	Impact Study
Ramirez and Cheng [1]	General	Collection of adaption patterns for self-adaptation expertise reuse.	News Web Server	No
Abuseta and Swesi [2]	General	Collection of design patterns for self-adaptive systems based on IBM MAPE-K multiple control loop issues.	E-learning Web Server	No
Berkane et al. [3]	General	Collection of design patterns for developing policies for self-adaptive systems at multiple levels of abstraction.	Smart Home System	No
Lakhani and Faisal [4]	Embedded Systems	Collection of design patterns with focus on software development for embedded applications.	No	No
Said et al. [5]	Embedded Systems	Propose five patterns to model a self-adaptive system, comprising the Monitor, Analyzer, Decision-Making, Acting and Assessing patterns.	Object Tracking and Resource Allocation Control Engine	No
Amorim et al. [6]	Embedded Systems	Systematic pattern-based approach that interlinks safety and security pattern engineering workflow.	Automotive System	No
This Paper	MPSoC	MPSoC middleware based on collection of Design Patterns and best practices of object-oriented programming for embedded systems, with focus on MPSoC domain.	Homogeneous MPSoC	Applications Execution Time and Memory Footprint

the applicability of the design patterns implemented in the environment, they present some case studies through an e-learning system. Furthermore, Berkane et al. [3] present an approach based on design patterns for developing policies for self-adaptive systems at multiple levels of abstraction. Such system considers feedback loops modeled in a modular way, and evaluates the execution in a smart home case study scenario. Although these works [1–3] present innovated approaches to achieve reusable design, they are designed with the focus on particular software development which comprises specific resource constraints.

Recent works propose the use of design patterns along with adaptive techniques to address the hard design constraints of embedded systems. Lakhani and Faisal [4] present a review regarding the evolution of design patterns developed for building architectures to diverse applications with a special focus on software development for embedded systems. Aiming to achieve performance and to cover real-time constraints, Said et al. [5] propose five design patterns used to model a loop-based self-adaptive embedded system. Further, Amorim et al. [6] present a systematic design patterns-based approach that interlinks safety and security patterns, considering an automotive use case scenario. All these approaches consider embedded systems and target particular designs that cover specific design constraints. However, none of them focus on embedded multiprocessor system-on-chip platforms. While these platforms provide parallel capabilities, the constraints include all embedded design restrictions, even more strict, increased by the complexity of management of multiple resources and applications.

Table I shows a comparison between the proposed approach and related works, emphasizing the performed impact at each work in the respective domain. This paper is the *first one* to present a middleware designed using design patterns and object-oriented languages aiming to improve software modularity and portability in the MPSoC domain.

III. BEST PRACTICES IMPLEMENTED IN THE PROPOSED MIDDLEWARE

The usage of design patterns allows the reuse of established solutions for known problems. In this section, we review approaches that employ or propose the use of design patterns to the MPSoC domain (Sec. III-A). Adding, we present some best practices in object-oriented programming for embedded systems (Sec. III-B) and details about the programming model implemented in the proposed middleware (Sec. III-C).

A. Collected Design Patterns

Aiming design flexibility, the *Container* design pattern comprises a holder object that stores a collection of other objects (its elements). Containers are implemented as class templates, supporting several element types. We have used two categories of containers: Sequence Container and Associative Container. Sequence Container stores objects and its elements in a strict linear order, providing methods for accessing them [9]. Examples of Sequence Container implementations are List, Queue and Deque. Associative Container stores objects based on keys (indexes), differing from sequence container since it does not provide insertion at a specific position [9]. Examples of Associative Container implementations are Map, Multimap, Flat-map and Flat-multimap. We use the Queue container to implement the Message Buffer component and the Flat-map and Flat-multimap containers to implement the managed topic tables in the proposed middleware (see Fig. 1).

Regarding design patterns for self-adaptive systems, the *Factory* design pattern [1] allows the decoupling of high-level elements (e.g., monitors, decision makers and actuators) from those elements that are target-dependent (e.g., processing elements and other low-level hardware/software components). This design pattern creates a standard interface that can be invocated in order to, for example, require information of a distributed monitoring infrastructure. We use the Factory design pattern to implement the Sensor/Decision Maker/Actuator interfaces detailed in the Sec. IV-B. The *Broker* design pattern [10] decouples the communication between the communicating elements in a publish-subscribe system. The *Observer* design pattern [10] defines a one-to-many dependency in a system with multiple both monitoring and actuation services. With it, all dependent objects are notified about a changing of a state in an object under observation. We use the Broker and Observer design patterns to implement the publish-subscribe programming model, detailed in the Sec. III-C.

Regarding software modularity, the *Hardware Abstraction Layer* (HAL) pattern [11] abstracts the underlying hardware/-

software structure from the rest of the system by implementing a driver with an abstract interface. We use the HAL pattern to enable portability in the proposed middleware.

Dynamic memory allocation is another issue in environments with restrict memory size. The *Fixed Memory Allocation* technique [11] allows the static allocation of the memory, with its maximum size defined at compilation time, avoiding unexpected behavior at runtime. We use this technique to improve runtime predictability and reduce the software code size when the dynamic allocation and adjacent library are used to compile the source code.

B. Selected Programming Language

Most of the operating systems or middlewares for embedded systems use C programming language because of the runtime efficiency and the high availability of compilers for a wide range of processors. Although design patterns can be implemented in a non-object oriented programming language [12], it leads to an awkward code, difficult to maintain. Recently, there have been efforts to use C++ in embedded systems. Most recent versions of C++, such as C++11[4] and C++14[5] standards, have enhanced features like type traits, operator overloading, static assertion, constant expression and concurrency support, enabling part of the C++ language support for embedded systems. The use of C++ in embedded systems with severe memory limitation (about tens or few hundreds of KBytes per processor) requires the use of techniques and best practices to reduce the code size generated by the compiler on the target platform. Following, we present some of these techniques applied to the proposed middleware. The impact of using these techniques is demonstrated in the Sec V-B2.

1) Placement new: The default *new* operator in C++ allocates memory in the kernel heap area and constructs an object in the allocated memory in execution time. This approach is usually not suitable for embedded systems. The *new* operator can cause unpredictable behavior in the lack of available heap memory space. Limiting the object's maximum allocation space to a fixed amount of memory at compile time is an alternative approach. Besides that, the default *new* operator increases the code size generated by the addition of all inherent methods of this operator, such as *malloc*, *mallocr* and *free*. *Placement new* approach [13] reimplements the *new* operator passing a pre-allocated memory area pointer and building the object in the given memory at compile time.

2) Avoid Exception Handling: Exception C++ feature adds to the code a large number of functions even when the exception feature is not used. So, in addition to not explicitly use exception handling in the code, it is advisable to include "*-fno-exceptions*" in the compiler options to disable this feature.

3) Compiler Optimization Options: The GCC and G++ compilers provide a set of options to control sorts of optimization. When used, they attempt to improve the performance and/or code size. The available set of optimization options depends on the target and how the compiler is configured. For example, when the primary goal is small memory size, the compiler could be instructed to optimize for size using the "*-Os*" flag in the compilation command. Other options are "*-O1*", "*-O2*" and "*-O3*" to performance optimization in exchange, generally, for large code size.

4) Embedded Template Library: The Standard Template Library[6] offers a set of well-tested design patterns implementations. However, it does not fit well in platforms with limited resource requirements. The Embedded Template Library[7] (ETL) is a worthwhile alternative designed for environments with restrict memory resources, since it provides containers with fixed capacity and static memory allocation. In the middleware presented in this paper, we use the following ETL containers patterns: *queue*; *map*; *multimap*; and its alternatives aiming size memory optimization - *flat_map* and *flat_multimap*.

C. Used Programming Model for Distributed Memory

There are several programming models for systems based on distributed memory, such as message passing, remote procedure call, client-server, and publish-subscribe (also called message queue). The publish-subscribe programming model is used in the proposed middleware for decoupling the application and system services development of the underlying hardware/-software infrastructure. In this model, the participants (nodes) communicate with each other by exchanging messages. The *broker* pattern is used to mediate the advertise and subscribe steps, decoupling, at the user level, the source of the messages from their destinations. A *subscriber* node manifests interest in a particular data or event identified by a *topic* (subscribe step). The node is notified when a message is published in this *topic* (publish step). The *publisher* node must register itself in the system as *topic* generator (advertise step). *Subscribers* interested in a *topic* receive notifications asynchronously.

IV. Proposed MQSoC Middleware

The middleware proposed in this paper, called Message-Queuing System on Chip (MQSoC), is based on the publish-subscribe programming model [8]. In addition, we are also proposing a new middleware structure based on an object-oriented approach improved with design patterns and programming best practices for embedded systems. The middleware structure also includes a Hardware/Software Abstraction Layer (HSAL), decoupling the middleware from the Operating System (OS) kernel and the hardware components.

Fig. 1 shows the proposed middleware structure and interfaces in a general MPSoC platform. From previous middleware implementation [8], we have only derived the publish-subscribe protocol phases. The remainder of the middleware structure has been fully redesigned. The gray blocks represent the modules implemented in this work. The white blocks represent hardware and software modules inherited from the base platform. The proposed middleware presents modules related to the publisher, subscriber and broker management. Containers represent data structures that store the topics data handled by each manager module. A message buffer, implemented through the Queue Container, retains the received messages before delivering them to the application layer.

[4]https://www.iso.org/standard/50372.html
[5]https://www.iso.org/standard/64029.html
[6]https://www.sgi.com/tech/stl/
[7]http://www.etlcpp.com

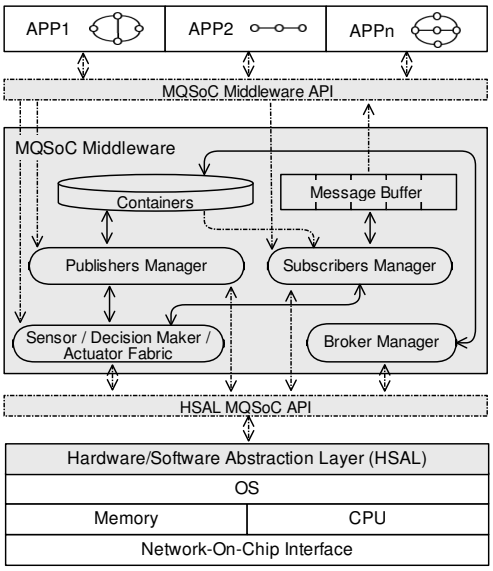

Fig. 1. Middleware structure and interfaces.

A. Platform Abstraction

The Hardware/Software Abstraction Layer (HSAL) presents a set of primitives that aims to facilitate the middleware portability in other platforms. It assures the middleware portability by the implementation of a specific driver for the target platform. The middleware code remains the same. The proposed HSAL has standardized functions to interface with the Kernel (i.e., to create, destroy, suspend, and resume system tasks), NoC (to write or read in the NoC interface) and MPSoC manager (i.e., to know if a node is the broker of the system).

B. Modular Software for Self-Adaptive MPSoC

The proposed middleware additionally provides an interface to manage self-adaptive services in the target platform. The Factory design pattern is used to model the set of sensor/monitor, decision maker and/or actuator components. The proposed interface allows to create different behaviors for these components and to call each one of them in a standard way. Fig. 2 shows the base interface classes (sensor/monitor, decision maker and actuator) and examples of derived classes (sensors of CPU temperature and buffer occupancy, an instance of a decision maker implemented through control theory, and an actuator in CPU clock). The Alg. 1 shows an example of sensors object creation in C++ using the *Placement New* technique (Sec.III-B1), and how the sensors are called by the middleware. Both sensors/monitors, decision makers and actuators objects can be statically or dynamically added to the middleware by using this interface. The Sec. V-C presents a case study of a self-adaptive application modeled using the proposed interface.

V. Experimental Setup and Results

This section presents the MPSoC platform used in all experiments (Sec. V-A), the performed experiments comparing the proposed middleware and the previous middleware implementation (Sec. V-B), and the self-adaptive application case study (Sec. V-C).

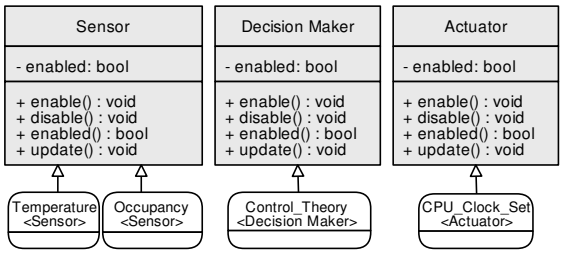

Fig. 2. Sensor, decision maker and actuator base classes and derivations.

Algorithm 1 Example using two sensors objects.

```
Sensor* Sensors[2];
size_t sensorLoc[2][SIZE_OF_SENSOR_OBJECT];
Sensors[0] = new (sensorLoc[0]) (Temperature);
Sensors[1] = new (sensorLoc[1]) (Occupancy);
...
for {int i=0;i<=1;i++}
    if (Sensors[i]->enabled())
        Sensors[i]->update();
```

A. MPSoC Platform

Fig. 3 illustrates the adopted case study platform composed of a 4x4 NoC-based MPSoC platform, with homogeneous processing elements (PEs) organized in clusters of 2x2 size. Each PE includes a Cortex-M4F processor, private random access memory (RAM) addressed to store the system (kernel+middleware) and applications, network interface, DMA and router. Each PE runs an extended FreeRTOS kernel independently, which uses cluster-based distributed management with dynamic task mapping feature [14].

The proposed middleware was incorporated into this platform through a specific HSAL implementation for the FreeRTOS kernel. The MPSoC hardware infrastructure was described using OVPSIM APIs[8] by Imperas, which provides an instruction accurate simulation framework.

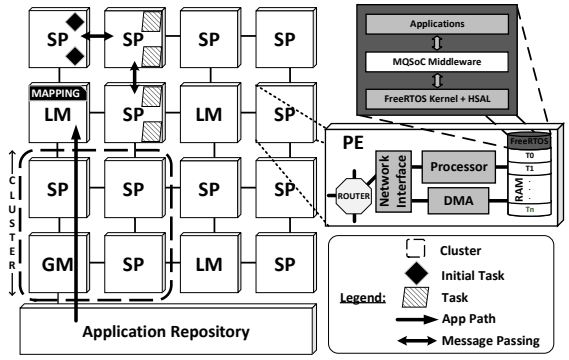

Fig. 3. Case Study MPSoC 4x4 platform instance (adapted from [14])

B. Middleware comparison

This section presents the evaluated applications benchmark and results regarding the middleware comparison against the previous middleware [8].

1) Evaluated Applications Benchmark: In order to compare the proposed middleware with the previous middleware

[8]http://www.ovpworld.org/technology_ovpsim

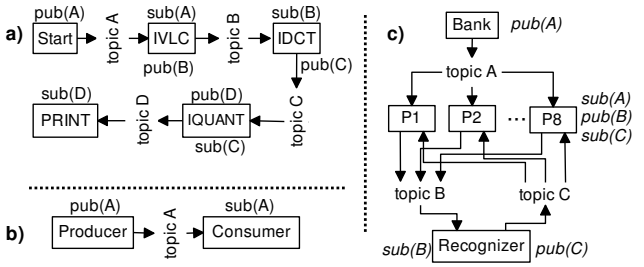

Fig. 4. Task graph of the a) MPEG, b) PROD-CONS, and c) DTW applications.

implementation [8], we have used a benchmark composed of the MPEG, Producer-Consumer (PROD-CONS) and DTW (Dynamic Time Warping) applications. Fig. 4 shows these applications represented through a task graph using the publish-subscribe programming model. In the Fig. 4, a directed arrow between two tasks (blocks in the figure) means that the first task sends data to the second one. A message encapsulates the data transferred between two tasks with a maximum payload size of 512 bytes in the experiments. The number of exchanged messages by an application depends on the workload data configured at compile time, in the test scenario.

2) Results: We compare two equivalent implementations of the middleware: a previous middleware implemented using C programming language [8]; and the C++ based middleware proposed in this paper (respectively C MIDD and C++ MIDD in the Fig. 5). Both implementations use the same base platform and programming model. The evaluated key metrics are memory footprint and application execution time. The used compilers are the *arm-none-eabi-gcc* and *arm-none-eabi-g++*, version 4.9.3. All scenarios use a single-cluster 4x4 MPSoC, with each PE executing a single task. Tasks are mapped in the same PE in both C and C++ evaluation scenarios.

The first experiment evaluates the achieved memory footprint size (kernel + middleware) of the C++ middleware implementation by using each technique and best practices for embedded C++, as presented in Section III-B. The Tab. II shows the footprint size of the initial version with no optimization (first line) and the footprint size achieved after using each technique, in the order in which they appear in the table. The total memory footprint size achieved in the final version of C++ the middleware is 17.03KB, being 11.06KB related to the kernel size and 5.97KB related to middleware size. Compared to the C-based middleware, the proposed C++ middleware represents an overhead of 1.37 KB (8.7%).

The second experiment evaluates the application execution time measured through a timing model [14] that capture the executed instructions for each processor, generating an

TABLE II. TOTAL MEMORY FOOTPRINT (KERNEL+MIDDLEWARE) IMPROVEMENT

Version	Footprint (KB)	Reduction
No optimization (INITIAL VERSION)	106	-
+ Using "-Os"	75.29	29%
+ Using "-fno-exceptions"	75.01	0.4%
+ Replacing Map/Multimap by Flat-map/Flat-Multimap	72.32	3.6%
+ Using "Placement new" (FINAL VERSION)	17.03	76.4%

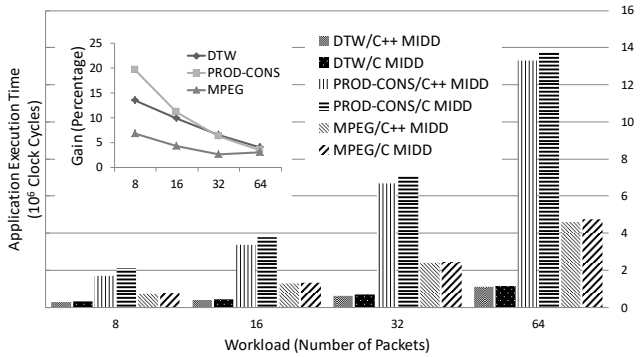

Fig. 5. Application execution time for C and C++ scenarios.

execution time from total executed instructions. For this experiment, we use the final version (last line in Tab. II) of the C++ middleware, which incorporates all cited optimizations. The execution time of both DTW, MPEG and PROD-CONS applications were evaluated ranging the workload data from 8 to 64 packets. Fig. 5 shows the results for each scenario. In all the simulated scenarios the C++ middleware presents better application execution time. The Fig. 5 also shows the percentage of gain obtained in the C++ middleware implementation compared with the C implementation (minor graph). It reduces the execution time ranging from 4% to 13.4% in DTW, from 3.4% to 19.5% in PROD-CONS, and from 3% to 6.7% in MPEG, depending on the workload. The gain decreases when the workload increases because the highest gain is obtained on topic advertise/subscribe steps, which occurs at the beginning of the application execution. In these steps, the use of containers reduces the time spent with the insertion and search of objects in the managed topic tables.

C. Self-Adaptive Application Case Study

This section shows an experiment performed to demonstrate the feature provided by the proposed middleware to manage self-adaptive services/applications.

1) Evaluated Application: For this experiment, the Producer-Consumer application was modified to allow an adaptive behavior on the data injection by the producer tasks. The application is composed of one or more producer tasks and one consumer task. The Fig. 6 shows a scheme of the self-adaptive application. The producer tasks inject data messages (of fixed size) in a configurable time interval (delay), and a local *actuator* (DataInjectRate) adapts the injection rate changing the delay at runtime. The actuators are commanded by the *decision maker* present in the consumer task. The

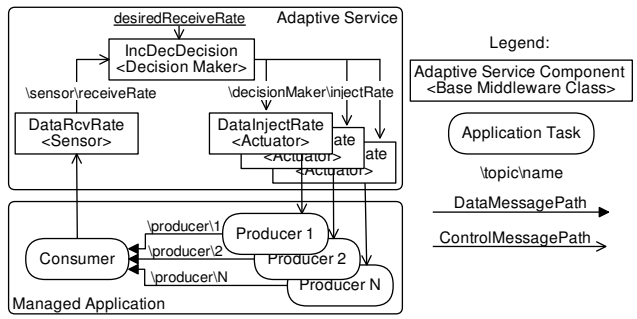

Fig. 6. Self-adaptive Prod-Cons application modelled using the middleware.

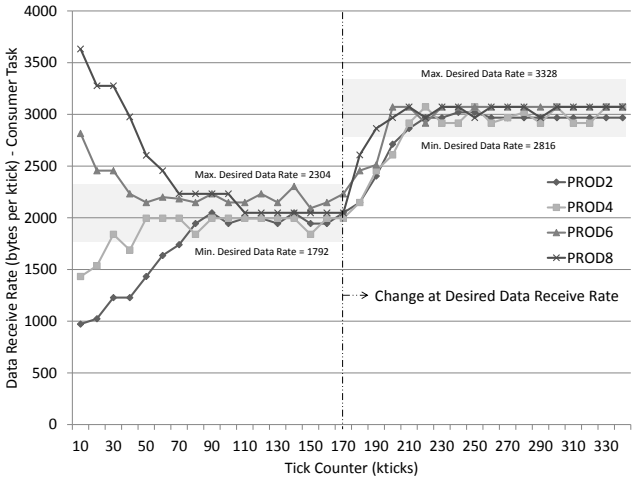

Fig. 7. Data receive rate adaptation at consumer task.

decision maker (IncDecDecision) monitors the received data rate (by subscribing to the topic "\sensor\receiveRate") and decides whether this rate should be increased, decreased or if nothing should be done. The decision is published in the topic "\decisionMaker\injectRate". Since the producer tasks have subscribed to this topic, they receive the decision messages, and the actuator configures the new delay. The actuation heuristic performed in the publisher tasks on the configurable delay allows to increase or decrease by 10% the delay depending on the decision message sent by the consumer task. After a period of stabilization, it is expected that the consumer task receives data messages at the *desired receive rate*. The *desired receive rate* represents the application requirement which must be achieved at runtime, expressing the adaptability of the system.

2) Results: Fig. 7 shows the results of this experiment. We have analyzed four scenarios, ranging the number of producer tasks (2, 4, 6 and 8) in each scenario (PROD2, PROD4, PROD6 and PROD8, respectively). For this experiment, we define a tick time of 1000 clock cycles. The consumer task performs a round of decision at every 10 ticks (10000 clock cycles). We defined an initial *desired receive rate* offset in the consumer task from a minimum and maximum values of 1792 and 2304 bytes per tick, respectively). Each producer task begins the data injection at an initial rate of 512 bytes per tick, defined at design time. The results show that the data receive rate in the consumer task is stabilized after around 80 ticks, reaching the desired rate offset in all scenarios. At the instant 170, we change the *desired receive rate* offset from 2816 to 3328 bytes per tick in order to show the application adaptability at runtime. As expected, the data receive rate is stabilized after instant 220.

VI. Conclusions

This paper presented a middleware to assist the development of self-adaptive management services and applications for MPSoCs. The proposed middleware is heavily based on object-oriented practices and design patterns to deliver platform abstraction, modular design, and decoupled distributed programming model. Despite the common sense in the field of embedded systems, where C programming language is predominantly used, this paper showed that modern C++ compilers along with best practices in object-oriented programming are competitive in term of memory footprint and performance. The performed experiments demonstrate that the proposed middleware is well tailored to MPSoC domain since it presents low memory usage and improved applications execution time. An additional experiment shows the applicability of the proposed middleware to design self-adaptive applications in the MPSoC domain. Future works include evaluating the modeling and implementation of emerging self-adaptive services aiming to satisfy application requirements while at the same time operating costs and energy-efficiency are optimized.

Acknowledgment

Thanks to Imperas Software Ltda. and Open Virtual Platforms for support and access to their models and simulator. Jean Carlo Hamerski is supported by CNPq and IFRS.

References

[1] A. J. Ramirez and B. H. Cheng, "Design patterns for developing dynamically adaptive systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2010, pp. 49–58.

[2] Y. Abuseta and K. Swesi, "Design patterns for self adaptive systems engineering," *Int. Journal of Software Engineering & Applications (IJSEA)*, vol. 6, no. 4, 2015.

[3] M. L. Berkane *et al.*, "A modular approach dedicated to self-adaptive system," *Lecture Notes on Software Engineering*, vol. 3, no. 3, p. 183, 2015.

[4] F. Lakhani and N. Faisal, "Design patterns-from architecture to embedded software development," *Int. Journal of Computer Science Issues*, vol. 12, no. 1, p. 146, 2015.

[5] M. B. Said *et al.*, "Design patterns for self-adaptive rte systems specification," *Int. Journal of Reconfigurable Computing*, vol. 2014, p. 8, 2014.

[6] T. Amorim *et al.*, "Systematic pattern approach for safety and security co-engineering in the automotive domain," in *Int. Conference on Computer Safety, Reliability, and Security*. Springer, 2017, pp. 329–342.

[7] D. C. Schmidt and F. Buschmann, "Patterns, frameworks, and middleware: their synergistic relationships," in *Software Engineering, 2003. Proceedings. 25th Int. Conference on*. IEEE, 2003, pp. 694–704.

[8] J. C. Hamerski, G. Abich, R. Reis, L. Ost, and A. Amory, "Publish-subscribe programming for a noc-based multiprocessor system-on-chip," in *IEEE Int. Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[9] D. A. Alonso *et al.*, *Dynamic Memory Management for Embedded Systems*. Springer, 2015.

[10] S. Tarkoma, *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012.

[11] V.-P. Eloranta *et al.*, "Patterns for distributed embedded control system software architecture," *Tampere University of Technology. Report 2*, 2009.

[12] B. P. Douglass, *Design patterns for embedded systems in C: an embedded software engineering toolkit*. Elsevier, 2010.

[13] K. Guntheroth, *Optimized C++: Proven Techniques for Heightened Performance*. " O'Reilly Media, Inc.", 2016.

[14] G. Abich *et al.*, "Extending freertos to support dynamic and distributed mapping in multiprocessor systems," in *IEEE Int. Conference on Electronics, Circuits and Systems (ICECS)*, 2016, pp. 712–715.