POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

DESIGN AND IMPLEMENTATION OF AN HOT-SWAP MECHANISM FOR ADAPTIVE SYSTEMS

Relatore: Prof. Marco Domenico SANTAMBROGIO

Tesi di Laurea di:

Filippo SIRONI

Matricola n. 734456

Anno Accademico 2009–2010

Alla mia famiglia

Ringraziamenti

In seguito alla conclusione di questo lavoro vorrei ringraziare molte persone per il supporto che mi hanno dato in questi anni di studio. Alcuni devono essere ringraziati per il supporto morale mentre altri vanno ringraziati anche per il supporto tecnico e professionale.

Prima di tutto voglio ringraziare i miei genitori, Renato e Ornella. Sembra banale dirlo ma è grazie a loro fino ad ora sono riuscito a realizzare tutto quello che mi ero preposto, mi hanno sempre spronato per fare in modo che facessi sempre del mio meglio. Mi hanno dato la possibilità di frequentare il Politecnico di Milano per 5 lunghi anni e, come se non bastasse, mi hanno permesso di frequentare la University of Illinois at Chicago, un'esperienza al di fuori dell'Italia che ritengo molto importante. Un ringraziamento particolare per mia sorella, Roberta (Beba), una persona a cui devo molto, indubbiamente un esempio da seguire e che ha mi ha sostenuto più di tutti in questi 5 anni di università. Un ringraziamento anche a tanti altri parenti, purtroppo non posso citarli tutti.

Un sentito ringraziamento a tutti i miei amici, a tutti i miei compagni di università ed a tutti i ragazzi del Laboratorio di Microarchitetture (il Micro). Con alcuni di voi sono cresciuto, con altri ho condiviso il periodo universitario a Milano (qui non posso proprio citarvi tutti... ma quanti siamo?!) ed a Chicago (in particolare Mario e il Villa, non ce la siamo cavata male in casa, vero?!) ed ovviamente dei bellissimi momenti al di fuori dell'università. Indimenticabile!

Finalmente è arrivato il momento per un grandissimo ringraziamento a Marco D. Santambrogio (il Santa), relatore ed amico, che ha fornito un prezioso aiuto per quanto riguarda tutto il mio lavoro di ricerca; se prima speravo di continuare a lavorare con lui ora ne ho la certezza, passeremo ancora degli anni insieme. Un ringraziamento particolare anche per tre amici e colleghi: a Marco Triverio (Trive), con lui ho condiviso lo sviluppo della prima parte del mio lavoro di tesi e parecchie ore in laboratorio oltre che 5 anni fantastici; a Luca Rocchini (Rocco o Tifone che dir si voglia), un'infinita fonte di consigli nonché enciclopedia portatile del Micro; ed infine un grande ringraziamento per Andrea Cuoccio (Andre o Cuoch che dir si voglia), che mi ha aiutato in questi ultimi mesi di lavoro.

Ancora una volta, un sentito ringraziamento a tutti voi.

Filippo

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AMD	Advanced Micro Devices
API	Advance Programming Interface
ABI	Application Binary Interface
BORPH	Berkeley Os for ReProgrammable Hardware
CO	Clustered Object
COID	Clustered Object Identifier
COR	Clustered Object Reference
COS	Clustered Objects System
CPU	Central Processing Unit
DDR	Double Data Rate
DES	Data Encryption Standard
DLL	Dynamic-Link Library
EC2	Elastic Compute Cloud
FAT	File Allocation Table
FPGA	Field Programmable Gate Array

GCC GNU Compiler Collection

GNU GNU is Not UNIX

GPGPU General Purpose Graphics Processing Unit

GPL General Public License

GPP General Purpose Processor

GPU Graphics Processing Unit

GTT Global Translation Table

HDL Hardware Description Language

HPC High-Performance Computing

IBM International Business Machines

ICAP Internal Configuration Access Port

IPC Inter-Process Communication

IPCM IP-Core Manager

IP-Core Intellectual Property Core

I/O Input/Output

KFS K42 File System

KST Kernel Symbol Table

LKM Loadable kernel module

LTT Local Translation Table

MAC Media Access Controller

MIMD Multiple Instruction Multiple Data

MIPS	Microprocessor without Interlocked Pipeline Stages
MPMC	Multi-Port Memory Controller
NFS	Network File System
NUMA	Non-Uniform Memory Access
ODA	Observe, Decide, Act
OTL	Organic Template Library
OTT	Object Translation Table
PC	Personal Computer
PCI	Peripheral Component Interconnect
PLB	Processor Local Bus
QoS	Quality-of-Service
RCU	Read Copy Update
RISC	Reduced Instruction Set Computer
RSoC	Reconfigurable System-on-Chip
SDRAM	Synchronous Dynamic Random Access Memory
SHA1	Secure Hash Algorithm 1
SMMP	Shared-Memory symmetric Multi-Processor
SO	Shared Object
SoC	System-on-Chip
SPMD	Single Program Multiple Data
STAPL	Standard Template Adaptive Parallel Library

STL Standard Template Library

TID Thread Identifier

UART Universal Asynchronous Receiver-Transmitter

USB Universal Serial Bus

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

VHSIC Very High Speed Integrated Circuit

XUPV2P Xilinx University Program Virtex-II Pro

XUPV5 Xilinx University Program Virtex-5

Summary

Nowadays computing systems can be seen as aggregates of complex computer architectures and software systems, spanning from grids including thousands of geographically distributed computing systems (*i.e.,* Amazon Elastic Compute Cloud (EC2)), to small and specialized embedded computing systems (*i.e.,* mobile devices). Our desktop computers are now equipped with multi-core processors, each of each may go up to 6 cores (*e.g.,* Intel Core i7 series and Advanced Micro Devices (AMD) Phenom X6 series), we are provided with many-core architectures (*i.e.,* NVIDIA and AMD Graphics Processing Units (GPUs)), and even our mobile devices are powered by extremely powerful System-on-Chips (SoCs) (*e.g.,* NVIDIA Tegra and Apple A4). We achieved such a complexity in the infrastructure, no matter what kind of computer architectures our computing system is based on, because of the increasing demand for performance, reliability, and for the need of always varying services.

We believe that in order to overcome the limits deriving by the increasing complexity of modern computing systems, one possibility is to adopt self-aware adaptive computing systems. Self-aware adaptive computing systems are capable of adapting their behavior and resources according to changing internal and environmental conditions and demands. This allows them to automatically find the best way to accomplish a given goal with the resources at hand. This capability would benefit the full range of computing systems, from small devices to grids of computing systems. Although

such a computing system may seem rather far fetched, we believe that basic semiconductor technology, computer architecture and software system have advanced to the point that the time is ripe to realize such a computing system.

In this wide scenario reconfigurable hardware may play an important role too, just like it does for some CRAY High-Performance Computing (HPC) solutions. Reconfigurable computing systems are capable to change their functionalities by dynamically adding or removing components. This enables hardware and software to adapt to changes dynamically at run time. Moreover, the increasing amount of programmable logic provided by modern Field Programmable Gate Arrays (FPGAs) makes it possible to configure multiple hardware components on the same device. This approach is reinforced by dynamic reconfiguration, which allows a single portion of the device to be reconfigured with another hardware component, without affecting the remaining portion of the device.

This thesis aims to define and implement self-aware adaptive computing systems characterized by the ability to modify their behavior and composition depending on their state and on the environment they work in, using hot-swap mechanisms implemented in both libraries and operating systems. We think hot-swap mechanisms, which allow the adoption of the best suiting implementation of a certain functionality according to a set of constraints (*i.e.,* goal, internal and environmental conditions), are fundamental for self-aware adaptive computing systems. The proposed solution consists in developing a sample self-aware adaptive library, following the proposed methodology, that enables a transparent self-aware adaptive behaviors for user mode applications running on operating systems based on the Linux kernel. This thesis work further proposes the adoption of an enabling technology: the stacked implementation of Loadable kernel modules (LKMs), to allow adaptation through hot-swap directly inside the

Linux kernel.

The remainder of this dissertation is structured as follows. A brief introduction over self-aware adaptive computing systems is provided in Chapter 1. An analysis of existing solutions based on either adaptive code or online reconfiguration is provided in Chapter 2. The definition of the proposed methodology and enabling technology for building self-aware adaptive computing systems is discussed in Chapter 3, and additional details about their implementations are reported in Chapter 4. Chapter 5 provides a set of results which are validated to demonstrate the goodness of the proposed approach, and finally, Chapter 6 concludes the document by discussing some possible future developments.

Sommario

Oggigiorno i sistemi informatici possono essere visti come un aggregato di architetture di hardware e sistemi software che svariano dai sistemi "grid" che includono migliaia di sistemi informatici distribuiti in tutto il mondo (*i.e.,* Amazon Elastic Compute Cloud (EC2)) ai piccoli sistemi informatici dedicati (*i.e.,* dispositivi portatili). I nostri Personal Computer (PC) sono equipaggiati con microprocessori multi-core fino a 6 core (*e.g.,* la serie Intel Core i7 e la serie Advanced Micro Devices (AMD) Phenom X6), abbiamo a nostra disposizione architetture many-core (*i.e.,* le Graphics Processing Unit (GPU) di NVIDIA e AMD) e persino i nostri dispositivi portatili sono dotati di System-on-Chip (SoC) estremamente potenti (*e.g.,* NVIDIA Tegra e Apple A4). Abbiamo raggiunto un tale livello di complessità a livello di infrastrutture indipendentemente dal tipo di architetture hardware e sistema software in uso, e ciò è avvenuto principalmente a causa della crescente necessità di raggiungere elevati livelli prestazionali, di affidabilità ed alla necessità di un numero di servizi in continua crescita.

Riteniamo che una delle possibilità per superare i limiti derivanti dal sempre più elevato livello di complessità dei moderni sistemi informatici sia l'adozione di sistemi informatici di tipo "self-aware adaptive". I sistemi informatici di tipo "self-aware adaptive" sono in gradi di adattare il loro comportamento e l'utilizzo delle risorse assecondando i cambiamenti dei loro parametri interni, delle condizioni dell'ambiente in cui operano e delle richieste a cui sono sottoposti. Questo permette a tale classe di sistemi di

trovare in modo automatico il modo migliore di raggiungere l'obiettivo richiesto con le risorse disponibili. Questa capacità garantirebbe dei vantaggi a tutti i tipi di sistemi informatici, dai piccoli sistemi dedicati ai grandi sistemi distribuiti. Sebbene un tale sistema informatico possa sembrare difficile da realizzare, riteniamo che le moderne tecnologie riguardanti le architetture hardware ed i sistemi software siano avanzate fino ad un punto tale per cui la possibilità di implementare tali sistemi informatici sia elevata.

In questo ampio scenario i sistemi informatici basati su hardware riconfigurabili possono giocare un ruolo di prim'ordine come già fanno in alcune soluzioni High-Performance Computing (HPC) di CRAY. I sistemi informatici riconfigurabili sono capaci di modificare le funzionalità che espongono aggiungendo e rimuovendo dinamicamente i loro componenti. Questo permette sia all'hardware che al software di adattarsi ai cambiamenti mentre il sistema informatico è in esecuzione. Inoltre, la sempre crescente quantità di logica programmabile messa a disposizione dalle moderne Field Programmable Gate Array (FPGA) rende possibile configurare una moltitudine di componenti hardware sullo stesso dispositivo; questo possibilità è rafforzata dalla possibilità di riconfigurare dinamicamente le FPGA, cosa che permette a singole porzioni del dispositivo di essere modificate con l'aggiunta o la rimozione di altri componente hardware, senza che il funzionamento della restante porzione del dispositivo sia compromesso.

Questo lavoro di tesi si propone di definire una metodologia per implementare sistemi informatici di tipo "self-aware adaptive" caratterizzati dalla possibilità di modificare il loro comportamento e la loro composizione in base allo stato in cui si trovano ad all'ambiente in cui lavorano, utilizzando dei meccanismi di "hot-swap" implementati sia a livello di librerie che a livello di sistema operativo. Riteniamo che i meccanismi di "hot-swap", che garantisco la possibilità di adottare l'implementazione di una data funzionalità che meglio si adatta alle attuali condizioni di lavoro ed alle richieste (*i.e.,* obiettivi, condizioni interne ed esterne), siano fondamentali per i sistemi informatici di tipo "self-aware adaptive". La soluzione proposta consiste nell'implementazione di diverse librerie di tipo "self-aware adaptive", seguendo la metodologia proposta, che garantiscono la capacità di adattare il comportamento delle applicazioni che vengono eseguite su un sistema operativo basato sul kernel Linux. Questo lavoro di tesi propone inoltre l'adozione di una tecnologia: l'implementazione di Loadable kernel module (LKM) composti da più moduli, un'interfaccia ed un'implementazione in modo da introdurre un meccanismo di "hot-swap" direttamente all'internod del kernel Linux.

Questo documento è strutturato come segue. Nel capitolo 1 viene fornita un breve introduzione sulla storia e sull'evoluzione dei sistemi informatici di tipo "self-aware adaptive" partendo dalle basi biologiche che hanno fornito un primo impulso alla loro definizione da parte di IBM nel manifesto dell'"autonomic computing" [1]. Il capitolo prosegue descrivendo l'Observe, Decide, Act (ODA) loop e le caratteristiche di base di un sistema di tipo "self-aware adaptive". Il capitolo 1 termina con una breve introduzione sui sistemi riconfigurabili che, come vedremo, rivestono un ruolo rilevante nello scenario che questo lavoro di tesi si propone di indirizzare. Il capitolo 2 propone un'analisi delle soluzioni esistenti basate su codice adattivo oppure sulla riconfigurazione online per implementare sistemi informatici di tipo "self-aware adaptive"; in particolare vengono proposti un framework per la realizzazione di applicazioni di tipo "self-aware adaptive" ad un sistema operativo sperimentale che si basa fortemente sui concetti dell'"autonomic computing". La definizione delle due metodologie proposte per implementare sistemi informartici di tipo "self-aware adaptive" e delle tecnologie adottate per indirizzare i problemi dell'osservazione del sistema e dell'ambiente, della decisione e della reazione sono riportate nel capitolo 3. Ulteriori dettagli riguardanti l'implementazione di metodologie e tecnologie sono riportati nel capitolo 4. Il capitolo 5 fornisce i risultati raccolti su due sistemi informatici caratterizzati da notevoli differenze che dimostrano la bontà dell'approccio proposto sia in termini di risultati effettivi che in termini di applicabilità. In fine, il capitolo 6 conclude questo documento discutendo alcuni possibili sviluppi futuri.

Chapter 1

Introduction

This Chapter highlights a common problem that plagues modern computing system, the skyrocketing complexity. We think the solution to this problem is self-aware adaptive computing and this Chapter provides an overall introduction on this topic and on motivations behind this thesis work. This Chapter ends with an introduction on reconfigurable hardware which can play an important role in improving the effectiveness of self-aware adaptive computing systems.

1.1 Problem statement

Nowadays we are surrounded by a vast range of computing systems spanning from grids [2], including thousands of geographically distributed computing systems, to small and specialized embedded computing systems. We achieved such skyrocketing complexity and computational availability because of the increasing demand for performance and reliability.

Designing and developing applications is a non-trivial process. Leaving software architects and software developers with the task of both solving problems that may arise during this process and taking care of thus details depending on hardware architectures, compilers, and operating systems is not feasible. We think that due to: the complexity of hardware architectures and hence their computational power availability and the complexity of system software manage the "optimization" problem either at design time and at compile time is nearly impossible. We consider autonomic computing, and therefore the adoption of self-aware adaptive computing systems, as the right answer to this problem. We envision a solution where the computing system itself is in charge of handling this complexity at run time thanks to its self-aware adaptive capabilities.

1.2 Research context: self-aware adaptive computing

This work finds its natural location in the *self-aware adaptive* or *autonomic computing* field which is an information technology related field placed under the big umbrella of *autonomic or autonomous systems*.

When autonomic computing was "introduced" [1] the term autonomic has been deliberately chosen with a biological connotation. The autonomic nervous system monitors our heartbeat, checks our blood sugar level and keeps our body temperature close to 37 °C without any conscious effort on our part, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, tasks [3].

Given the previous example we can define an *autonomic system* as a system that operates by managing itself without external intervention even if it operates in changing environments.

Autonomic activities in a system are accomplished by taking an appropriate sequence of actions based on one or more situations perceived from the system itself and from the environment. This bring us to define a control-loop in which the system collects information from the environment and from its internals, takes an appropriate decision and finally acts accordingly. A lot of names are used to refer to this process; we generally

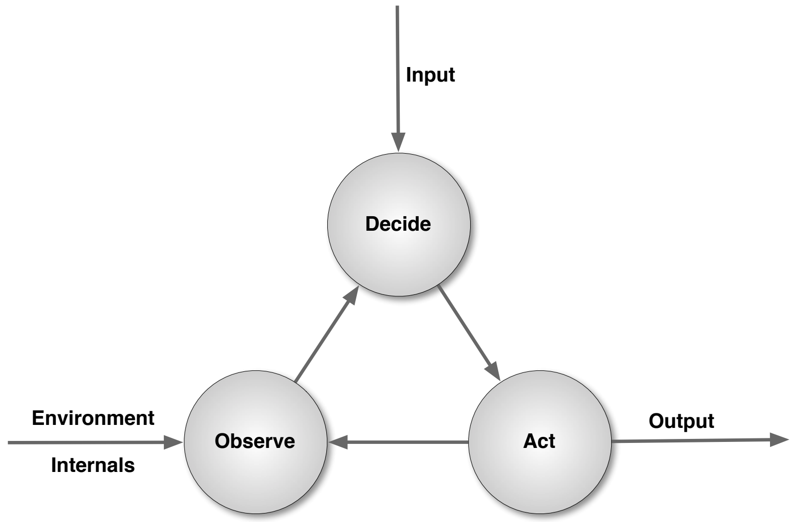


Figure 1.1: ODA loop

call this control-loop: the *Observe, Decide, Act (ODA) loop*. Figure 1.1 shows how the control-loop works.

Every autonomic system is based on three fundamental principles; *observation* which is done by the sensing sub-system to monitor both the environment and system's internals maintaining the *state* information; inherent to an autonomic system is the knowledge of *goal* which guides the logic sub-system to come up with an appropriate *decision* based on both the state and the external stimuli; finally the decision "translated" into a set of *actions* which are executed by actuators.

A widely adopted model for autonomic systems that summarize what has been told so far is shown in Figure 1.2.

Even though the purpose and thus the behavior of autonomic systems vary from system to system, every autonomic system must exhibit some properties, which are a minimum common denominator, to achieve its purpose. Autonomic systems are:

- *automatic*, which essentially means for an autonomic system to be able

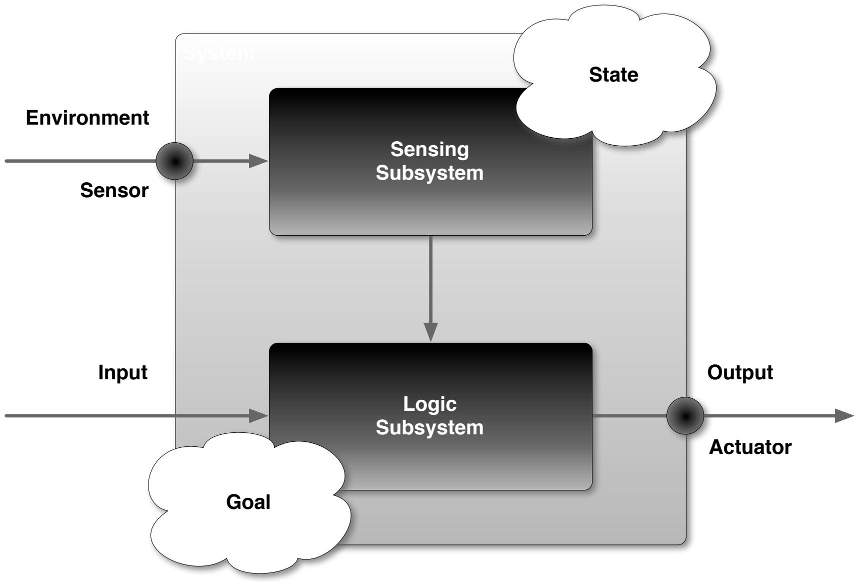


Figure 1.2: Autonomic system generic model

to self-control its internals. As such, an autonomic system must be a self-contained entity;

- *adaptive*, which essentially means for an autonomic system to be able to self-heal and self-configure (self-govern) its internals allowing the system to react to both long and short term changes;
- *aware*, which involves knowledge on both the autonomic system itself and on the context in which the autonomic system lives. Therefore, an autonomic system is self-diagnose with respect to its internals and it is context-diagnose since it monitors the environment in which it is immersed.

In a more computer science oriented fashion, we can define an *autonomic computing system* stating that the core principle of *autonomic computing* is to manage technology using technology itself in order to help addressing the rapid increase of complexity in information technology [3].

The idea is to avoid costly and time-consuming procedures and solve problems with minimal human intervention focusing ourself on higher-level tasks.

There are a lot of similarities between organic systems and autonomic computing systems. Both of them are made up of a myriad of interacting subsystems, which can be autonomic systems as well. On one side there are organs while on the other side integrated circuits and their total amount is comparable.

However, there is an important distinction between autonomic activities in organic systems and autonomic activities in computing systems. In the first case many of the decisions made by autonomic capabilities are involuntary while in the second case autonomic capabilities perform tasks that we - human beings - choose to delegate to the technology according to policies which can be either adaptive or fixed.

Leaving the more general field of autonomic systems and focusing on autonomic computing systems, what are the pillars of autonomic computing? How might autonomic computing systems work?

1.3 Autonomic computing pillars

The declared objective of autonomic computing is *self-aware adaptiveness*, the intent of which is to free human beings from the burden to care about operational and maintenance details of computers and to provide users with the best user experience in terms of both performance and availability.

Autonomic computing systems should adapt their internals upon changes in a many parameters, for example, the overall workload and demand, hardware and software failures (both innocent and malicious), power consumption, required performance, and so on. This is due to continuos monitoring done by the system itself. In a futuristic scenario a system could even fully upgrade itself if necessary, run a series of regression tests to ensure functional correctness and upon errors roll back to the previous safe checkpoint. All these operations should be done while the system is up and running in order not to compromise the availability.

According to International Business Machines (IBM) [1], which was a pioneer in autonomic computing, there are four different properties characterizing self-aware adaptiveness that must be implemented in order to claim a computing system belongs to the autonomic computing systems set. These four properties, sometimes called *self-* properties* [4], are:

- *self-configuration*, installing, configuring and integrating complex systems is a tough task which proves to be error-prone even for experts system administrators. The idea behind self-configuration is that autonomic computing systems should configure themselves automatically and dynamically following high-level policies which specifies what is desired and not how it is to be accomplished. Every new component should incorporate itself seamlessly by registering its capabilities so that other components know about them - going beck to the dualism between organic and computing systems, this process should be much like the addition of a cell inside an organism;

- *self-optimization*, computing systems have thousand of parameters that can be set in order to finely tune performance and this number tends to increase with each new release. Finding the right settings proves to be difficult even for humans. Continuing the parallelism between organic and computing systems, an autonomic computing systems should continually seek opportunities and ways to improve their own performance - just as muscles do through training. Sometimes self-optimization is called *self-tune* or *self-adjust*;

- *self-healing*, problem identifying and tracing in large, complex computing systems can take a considerable amount of time and a considerable human effort. By means monitoring process - which can deepened from a normal to a more verbose level - autonomic computing systems acquire more data to analyze, this way systems should find solution to heal problems themselves in case of both hardware and software failures. This property is strongly related to *self-diagnosis* and *self-repair*;

- *self-protection*, even though advance protection tools such as firewalls and introduction detection systems exist, they are difficult to configure since they implement low-level policies. Autonomic computing systems automatically defend against malicious attacks or failures maybe using early monitoring to anticipate threats and disasters[1]. Hence, a self-protecting computing system can operate reactively or proactively.

Self-* properties are somehow related to system's quality factors. Self-configuration impacts on maintainability, functionality and usability; self-optimization has strong implications in efficiency while both self-healing and self-protection have relationships with availability and reliability.

1.4 Motivations

So far we gave an introduction and a definition of autonomic computing and we highlighted few hints on what are the pillars of autonomic computing. Let's now focus on what are the real advantages and motivations behind autonomic computing and why we should prefer autonomic computing systems respect to simple computing systems.

[1]A disaster is a serious fault such that the system availability is compromised.

Computing systems dealing with changing environments normally require supervision from human beings to continue operate in all conditions. The amount of costly and time-consuming procedures needed to maintain those systems is somehow increasing due to the complexity of modern computing systems. Since there is and increasing demand for automaticity autonomic computing is born and is now widely considered the right approach.

With the introduction of the autonomic capabilities presented in Section 1.3, a system can improve:

- *performance*, the best resource management mechanisms and policies depends on many parameters. These parameters usually vary as the time goes on the system as to change both resource management mechanisms and policies in order to reach the optimal performance. For example, highly parallel data structures are need in multi-processor environments to achieve good performance while different data structures are usually needed in sequential environments;
- *customizability, extensibility*, the best performance can be reach through evolution. Hence, the system is meant to tailor itself according to its goals and with respect to the conditions in which it works. If the system reconfigure itself without augmenting its capabilities then the system is customized, if the system reconfigure itself augmenting its capabilities then the system is extended. To give and example, the ability to change the data structure used by an application, in favor of a more suitable one, is an example of customization while the introduction of a brand new search algorithm, which wasn't available, inside a software library is an example of extension;
- *maintainability, availability*, through customization the system can modify the way it works, however, the system itself must ensure the cor-

rectness after it changes the way it works. Moreover, the fact that the system changes the way it works must not decrease the system availability. Customization and extension can be used to maintain the system updating its components with newer releases;

- *security*, thanks to the fact that the system is capable to reconfigure itself while it is running it can also fix security holes through customization. Installing newer releases of components at every system level with the intention to fix security holes without interrupting the work is a clear advantage and yet a good motivation.

With improved performance, customizability, extensibility, maintainability, availability and security, an autonomic computing system reduces the effort and skills needed to use and *manage* the system. However, this simplification introduces a lot of complexity at design-level, hence the effort and skills needed by system programmers are much higher.

1.5 Reconfigurable hardware

For self-aware adaptive computing systems the ability to choose among available functionalities and maybe different implementations of the same functionality the best one to accomplish a certain task with respect to a set of constraints and conditions is fundamental. The use of reconfigurable hardware side-by-side to “static” hardware and software introduces an additional level of customization and hence even more freedom with respect to traditional solutions involving solely “static” hardware and software. This new degree of freedom provides the capability of introducing even more choices to self-aware adaptive computing systems.

The concept of *reconfigurable computing* has been around since the 1960 when Gerald Estrin, a computer scientist at the University of California, Los Angeles, proposed the concept of a computer architecture combining some of the flexibility of software with the high performance of hardware. The result was a computer consisting of a standard processor augmented by an array of *reconfigurable hardware* [5] where the first was in charge to control the behavior of the reconfigurable hardware. The latter is tailored to a specific task and as soon as the assigned task is completed the reconfigurable hardware can be changed to perform some other task.

The principal difference between reconfigurable hardware and programmable hardware (*e.g.,* microprocessor) is the ability of the first to be substantially modified at datapath level in addition to the control flow level. On the other hand, the main difference between reconfigurable hardware and custom hardware is the possibility for the first to be adapted at runtime.

Nowadays the most widespread type of reconfigurable hardware is the *Field Programmable Gate Array (FPGA)*. The FPGA configuration can be specified using a variety of languages, the two most famous are Hardware Description Languages (HDLs) (*i.e.,* Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) and Verilog).

FPGAs configuration capabilities allow a great flexibility and make it possible to create a vast number of different reconfigurable computing systems. These can vary from boards featuring minimal Input/Output (I/O) interfaces and a variable number of FPGAs to build System-on-Chip (SoC) and Reconfigurable System-on-Chip (RSoC) or workstation coming with FPGAs-based boards, mounted on standard busses such as the Peripheral Component Interconnect (PCI) or the PCI-Express logic busses, used as accelerators. Both this reconfigurable computing systems are particularly interesting for this work.

1.6 Outline of this dissertation

In this Chapter we proposed an overview on the fundamental concepts and the terminologies used. It is important to notice that sometimes other terminologies are used in place of autonomic computing systems; researchers often talk about *self-managing computing systems* or *self-adaptive computing systems* even though other researchers claim there are distinction [4]. In the remainder of this work various terminologies are used to best fit the specific argument, however, we usually refer to autonomic computing systems as *self-aware adaptive computing systems* which suits the overall work.

Chapter 2 shows how the concept of self-aware adaptiveness has been absorbed by contemporary computing systems and we will take a look at different kind of approaches which works at different levels. Chapter 3 describes the proposed approach to embrace self-aware adaptiveness highlighting those points in which it differs from the state-of-the-art. Chapter 4 discusses implementation details while Chapter 5 lists all the preliminary results obtained. Chapter 6 concludes, discussing observations and future works.

Chapter 2

State of the Art

As anticipated in Chapter 1, this Chapter presents *state-of-the-art* solutions implementing a self-aware adaptive behavior. In particular, two different philosophies to achieve self-aware adaptive computing are presented trying to highlight the advantages of one approach with respect to the other one. These paths to enable self-aware adaptive behavior in computing systems are: *adaptive code* and *online reconfiguration*.

2.1 Adaptive code: the framework for adaptive algorithm selection in STAPL

Writing portable applications that perform well on a vast set of platforms with variable input sizes and data types proves to be extremely difficult because of performance sensitiveness to the system architecture, internal state, and environmental conditions variability. One common approach to implement the self-aware adaptive behavior needed to address this complexity is adaptive code. The idea behind adaptive code is to provide either multiple implementations of the same functionality and then select the best available implementation within this set considering all the factors reported above or, as an alternative, design a single highly tunable implementation.

Adaptive code can be exploited at different levels and each of these level is more or less dynamic than the others. To give a practical example let's see what happen in portable software and the *Linux* kernel is a wonderful example of portable software since it currently runs on 22 different architectures (without considering all the sub-architectures available). There are plenty of aspects of an operating system kernel that strongly depends on the underline architecture (*e.g.,* Input/Output (I/O) ports mapping inside the main memory address space or inside an I/O-reserved address space). This means some portions of the code may vary and the build process is responsible to "choose" at compile time the correct portion of the architecture dependent code to be linked with the architecture independent code. The Linux kernel is a valid example in which a "static" approach to adaptive code is used since all of the possible choices are taken at compile time. In other scenario software built upon adaptive code have run time parameters that can be modified while the software is running, function calls within a Linux device driver can have different behavior and this behavior can be changed tuning parameters exposed by means of interfaces such as `sysfs`, `procfs`, or system calls (*i.e.,* `ioctl()`). In other situations software comes with multiple implementations of the same algorithm and/or with different data structures, and the best implementation to adopt can vary over time. For example, software using adaptive code may be combination of several individual algorithms, each designed for a particular workload and those optimized for a specific situation of both the internal state and the environment state.

It is important to notice that even though adaptive code is a "fast" way to achieve self-aware adaptive behavior hence helping to partially fulfill the vision previously discussed, it can actually fail to get the full autonomic computing vision we are willing to implement.

PetaBricks is a "young" research projects discussed in [6, 7] that can be consider a self-aware adaptive solution biased toward adaptive code. It is an implicitly parallel language and compiler, in which having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. In PetaBricks algorithmic choice becomes a first class construct of the language. Choices are provided in a way that also allows the compiler to autotune programs by making both fine-grained as well as algorithmic choices. Fine-grained choices include different automatic parallelization techniques, data distributions, algorithmic parameters, transformations, and blocking.

An interesting example of adaptive code solution is *Organic Template Library (OTL)* [8, 9]. OTL is a software library of common data structures and algorithms that programmers can use to make parallel programming easier. OTL applies bio-inspired adaptation strategies to auto-tune itself during program execution to optimize performance, parallelism, and power consumption so that programmers need not manually address these issues. OTL is patterned after the C++ Standard Template Library (STL)[1] in that OTL's interfaces for data structures and algorithms are similar. The OTL's interfaces present a sequential programming model but the underlying implementations are parallel. For suitable application domains, the sequential programming model provides a convenient abstraction that allows the programmer to ignore the complications of parallelism. Under the hood, OTL's implementations are very different from STL's. OTL components dynamically self-optimize in response to runtime conditions and performance feedback, and they adapt to environmental factors such as input data characteristics and the availability of system resources.

Another example of adaptive code solution, the one we consider as the

[1]The C++ STL is a software library included in the C++ *standard library* and it basically provides containers, iterators, algorithms, and functors - also called function object - to help programmers avoid rewriting common code.

state-of-the-art, is *Standard Template Adaptive Parallel Library (STAPL)*. STAPL is a parallel C++ library designed as a superset of the C++ STL, and is implemented using simple parallel extensions of C++ which provide a Single Program Multiple Data (SPMD)[2] model of parallelism also supporting recursive (nested) parallelism. Basically, STAPL provides at least one (hopefully more) sequential and parallel implementations essentially for containers, iterators, and algorithms. The choice among these different implementations is made adaptively on a performance model, statistical feedback, and current run time conditions [10]. STAPL is intended to possibly replace STL in a transparent manner allowing programmers to write portable applications scaling well on a vast variety of computing systems.

STAPL provides mechanisms for both programmer directed and autonomic algorithm selection. For example, programmers may want to use their "a priori" knowledge to force a certain algorithm proved to be efficient within a precise context.

If the algorithmic choice is left to STAPL then a more general framework studied to provide the self-aware adaptive behavior starts its tasks. This general framework is described in [11] and Figure 2.1 shows the major components and the flow of information within the framework. The general framework basically surround STAPL with two important components: (1) *data storage* (*i.e.,* data repository implemented with a database) to collect data retrieved by the framework installation benchmark and applications run time, (2) *machine learning* which works on the retrieved data to determine run time tests that will select among algorithmic options, which are the algorithm itself and those parameters meant to tune performances.

In fact, the installation procedure collects statistically useful information such as the number of processors and cores, caches size, and available

[2]SPMD is the most diffused parallel programming model and is derived from Multiple Instruction Multiple Data (MIMD); in SPMD tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.

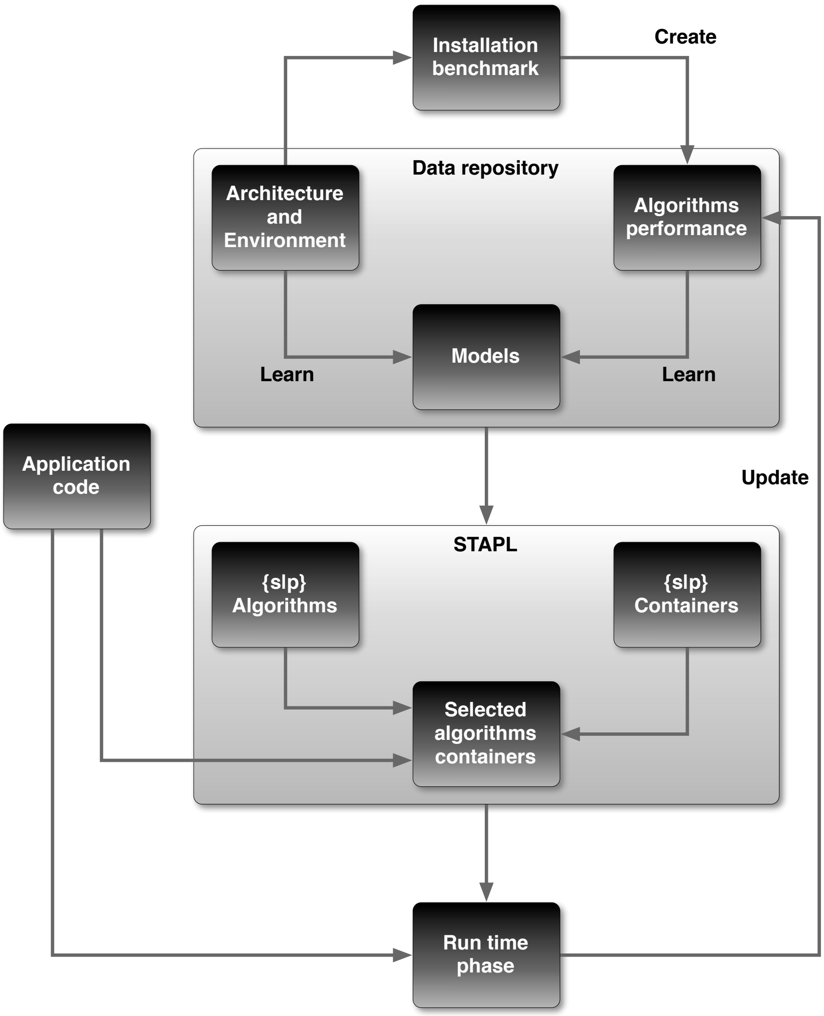


Figure 2.1: Adaptive algorithm selection framework

memory. This "static" information is stored in the data repository alongside with "dynamic" data collected by means of a set of benchmarks to evaluate the performance of available algorithmic choices which is further refined thanks to the data collected from applications run time. Next, machine learning techniques are used to analyze the information stored in the data repository and to determine those run time tests which are responsible to guide selection of both algorithms and tuning parameters.

Self-aware adaptive computing systems require real time data; it is therefore imperative for such computing systems to incorporate from the very beginning run time instrumentation for monitoring. Hence, applications using STAPL are instrumented to feed the data repository with the necessary information to refine subsequent algorithm selections.

2.2 Online reconfiguration: operating systems for reconfigurable hardware

During the last ten years a lot of effort has been spent to design and implement operating systems supporting reconfigurable hardware. Most of the existing reconfigurable computing systems include one or more General Purpose Processors (GPPs), either hard processors or soft processors[3], which are typically employed to execute the operating system and the software that manages the reconfiguration process [12]. Researchers focused on the adaptation of existing, widespread, and general purpose solutions such as the *Linux kernel* for embedded computing systems. In a reconfigurable computing systems, the most important feature an operating system should provide is the exploitation of the reconfigurable hardware devices by the

[3]A soft processor is a processor that can be totally implemented using logic synthesis and can be completely mapped on semiconductor devices such as Field Programmable Gate Arrays (FPGAs).

different processes running on the operating system.

In [13] an operating system for reconfigurable computing systems has been presented and discussed. The proposed approach is based on the concept of *hardware task*, which is an Intellectual Property Core (IP-Core) mapped on the area of the reconfigurable hardware device. The discussed operating system has not been implemented, but it is nevertheless an important starting point for the management of hardware functionalities, to exploit the multitasking capabilities offered by the parallel allocation of different hardware tasks on reconfigurable hardware devices. However, online reconfiguration can be exploited not only to map IP-Cores on the reconfigurable hardware device, but also to create reconfigurable computing systems in which hardware functionalities of any kind can be switched in and out, by means of mapping and unmapping them on reconfigurable hardware devices. An example of this solution, can be found in [14], where an online reconfigurable computing system based on the μ*Clinux* [15] operating system has been presented and evaluated to accelerate an computational intensive task by means of dedicated hardware.

Two similar platforms with support for online reconfiguration are presented in [16] and [17]. In both of them μ*Clinux* is used as the basic operating system which is extended in order to allow online reconfiguration by means of the support implemented within kernel modules. In particular, a device driver has been developed for the Internal Configuration Access Port (ICAP) [18] reconfiguration port which is used to physically modify a portion of the FPGA, the chosen reconfigurable hardware device. The device driver provides access to the ICAP through a character device (*i.e.*, `/dev/icap`), which allows the use of the reconfiguration port by means of the standard system calls (*i.e.*, `read`, `write`, and `ioctl`), without any knowledge on the low-level details of the ICAP. In addition to the device driver to pilot the ICAP, in [17] the operating system is further extended including another kernel module called IP-Core Manager (IPCM), which is employed to register new hardware functionalities, to load the corresponding device drivers, and to make the IP-Cores available for user mode processes. The solution presented in [19] can be considered as an improved version of the one discussed in [17]. These two works differ in the online management of the reconfiguration process. The latter work introduced the centralized reconfiguration manager to support and manage both external and internal online reconfigurations.

Another valuable solution, which is worthy of a mention, is *Berkeley Os for ReProgrammable Hardware (BORPH)* [20, 21]. BORPH is an operating system designed for FPGA-based reconfigurable computing systems. It is an extended version of the Linux kernel that handles FPGA resources as if they were Central Processing Units (CPUs). BORPH introduces the concept of *hardware process* (recalling the *hardware task* concept presented in [13]) which behaves just like a normal user process except it is an hardware design running on a FPGA. Hardware processes behave like normal software process. The BORPH kernel provides standard system services, such as file system access, to hardware processes, allowing them to communicate with the rest of the system easily and systematically. However, this approach does not provide the possibility to execute a partitioning of a given process in software functionalities and hardware functionalities, in addition to this, there is no automatic flow that is able to generate an hardware process from a high-level specification. Thus each change on the high-level specification of the problem has to be directly translated in a manual change of the low-level hardware description. Furthermore, the reconfiguration of an hardware process is encapsulated in the executable file so it is hard to completely exploit hardware re-use and it is therefore impossible to implement caching policies. Moreover, the resulting reconfigurable computing systems is somehow "static" since it is not allowed to choose at run time the most suitable hardware implementation of the hardware process that has to be deployed, for example depending either on the FPGAs availability or on the required performance because this choice is bounded at design time.

After discussing online reconfiguration solution for reconfigurable computing systems, next Section illustrates a solution implementing online reconfiguration, using software techniques, for traditional computing systems.

2.3 Online reconfiguration: the K42 operating system

Central to self-aware adaptive computing is the ability of a system to identify problems and to reconfigure itself in order to address them. We describe online reconfiguration as a technology that can address all autonomic requirements previously discussed and how online reconfiguration was successfully implemented in a research operating system.

First of all we need to discuss the very base of online reconfiguration: the framework behind each online reconfiguration process. Each online reconfiguration process can ben decomposed in ideally "atomic" steps which are reported below in the order they must be performed and described in [22, 23].

1. *Triggering the process*, the online reconfiguration of an object may be triggered from the object itself or from another object. Monitoring is a fundamental component for the triggering process. For example, an entity might use interposition first in order to acquire more information to understand the source of the problem (*e.g.,* lack of performance, detection of intrusion) and make a better decision (*i.e.,* if there are more than one candidate for the substitution the system should choose the one that suits better). The entity that takes the decision

can be the object to be replaced whenever it discovers it doesn't behave as it should or a third party object (*e.g.,* application) that want to customize the system behavior;

2. *choosing the new object*, sometimes the target is well-known, the initiator knows the name of the new object; in other occasions the substitute may be identified by a sub-set of its features, thus the system should expose some kind of facility allowing objects to publish their characteristics. If the initiator knows some of the characteristics of the new object it can query the system to extract all the candidates that satisfy the constraints;

3. *instantiating the new object*, it should be a straightforward task which requires no support from objects. This step is where the "real" online reconfiguration starts;

4. *getting the old object in a quiescent state*, a quiescent state needs to be established so that it is safe to transfer it. As one might expect the a quiescent state can be achieved when no one is executing the object code; there are many ways to reach such state, a simple one consists in acquiring a write lock on the object (while users are required to acquire a read lock to operate), the grant of such a lock means no one is using the object anymore. Being a simple technique, this method is also inefficient (*i.e.,* it does not scale on multi-processor systems);

5. *transferring the stable state*, once a quiescent state has been established it has to be transferred from the old object to the new one. This operation can be done with serialization or canonical forms, however, the use of an agreement protocol is indeed preferable;

6. *changing the references*, the whole set of references pointing to the old object needs to be change to point to the new object to allow users

to access the new implementation instead of the old one. Possible solutions to achieve this goal consists in implementing mechanisms similar to garbage collectors (which may result inefficient on multiprocessor systems) or distributed objects (*i.e.*, fragmented objects and partitioned objects [24, 25]);

7. *deleting the old object*, this is the last step and should be simple enough not to require support from objects.

Among these steps, getting a quiescent state, transferring it and changing all the references are the most challenging since they directly touch the state (of both the object and the whole system). Moreover, these challenging steps require direct support from objects that want to comply to the online reconfiguration capability.

Given this framework, we can claim online reconfiguration has four requirements [23]: first of all, components must have *well-defined boundaries*, second, components must be capable of being forced into a *quiescent state*, third, components must support *state transfer*, and fourth, it must be possible to *update every reference* to components.

2.3.1 Structure

Having seen an exhausting introduction on online reconfiguration, we are now ready to talk about those enabling technologies K42, a research operating system, has introduced to implement online reconfiguration and how these technologies work to fulfill autonomic requirements.

K42 is an open source research operating system willing to provide a well-structured kernel but maintaining performance as central concern. Some research operating systems embrace particular philosophies and follow them rigorously in order to fully examine their implications. Even though K42 has precise objectives the research group behind the development is ready to come at some compromises for the sake of performance.

K42 is currently developed by International Business Machines (IBM) while the core of K42 is based on a previous work of the *University of Toronto*. K42 is actually the third generation and its predecessors are the *Tornado Operating System*, which is the second generation, and *Hurricane Operating System*, which is the first generation.

K42 is a scalable operating system for both Shared-Memory symmetric Multi-Processor (SMMP) and Non-Uniform Memory Access (NUMA) 64-bit computing systems, which currently runs on PowerPC and Microprocessor without Interlocked Pipeline Stages (MIPS) platforms [26]. The main goals of K42 are [26, 27]:

- *scalability and performance*, K42 runs efficiently on a variety of multi-processor systems, it scales up well on large multi-processor systems and scales down well on small multi-processor or uni-processor systems;

- *adaptability*, K42 manages systems resources in a way that matches the changing needs of the running applications and that contributes to the autonomic behavior of the system;

- *customizability, extensibility and maintainability*, K42 open source nature guarantees a natural high degree of customizability; K42 support is extensible to additional hardware platforms - a porting to x86-64 is on going - or applications; upgrading the system with new components is simple without interruption in system services.

K42's features leverage on few important concepts that guided the whole development process:

- the use of modular object-oriented code. The object-orientation helped to develop a scalable operating system kernel since every system resource is managed by a "per-instance" object or set of objects. This wise choice guarantees applications the ability to best serve their needs which can vary as the time goes on. Autonomic capabilities are provided in order to swap "on the fly" these per-instance objects;

- the avoidance of centralized code and data structures favoring distributed code and data structures. Thanks to this decentralization the programmers can avoid to use global locks which usually degrade both performance and scalability;

- the transfer of system functionalities from the kernel to application libraries and to server processes according to the well-known *microkernel* [28] design. This choice helped in developing a more robust and maintainable operating system kernel.

K42 operating system kernel is structures on the well-known client-server model illustrated in Figure 2.2.

The kernel provides the memory management, process management, Inter-Process Communication (IPC) infrastructure, base scheduling, networking support and device support. Right above the kernel there are two other layers: the server layer and the application layer. The servers may include file servers like Network File System (NFS) or K42 File System (KFS)[4] [29], a name server, a socket server and a pipe server [26]. To avoid an heavy amount of context switch between the user mode and the kernel mode most of the communication facilities are implemented in application libraries (*e.g.*, the whole thread scheduling is done by a user mode scheduler linked into each process).

Communication paths between clients and servers are automatically generated by means a stub compiler which works on C++ classes augmented with decorations[5], these communication paths are also optimized and are attached with sufficient information to authenticate each call.

[4]A scalable file server.

[5]Annotations that extend the language for additional functionalities.

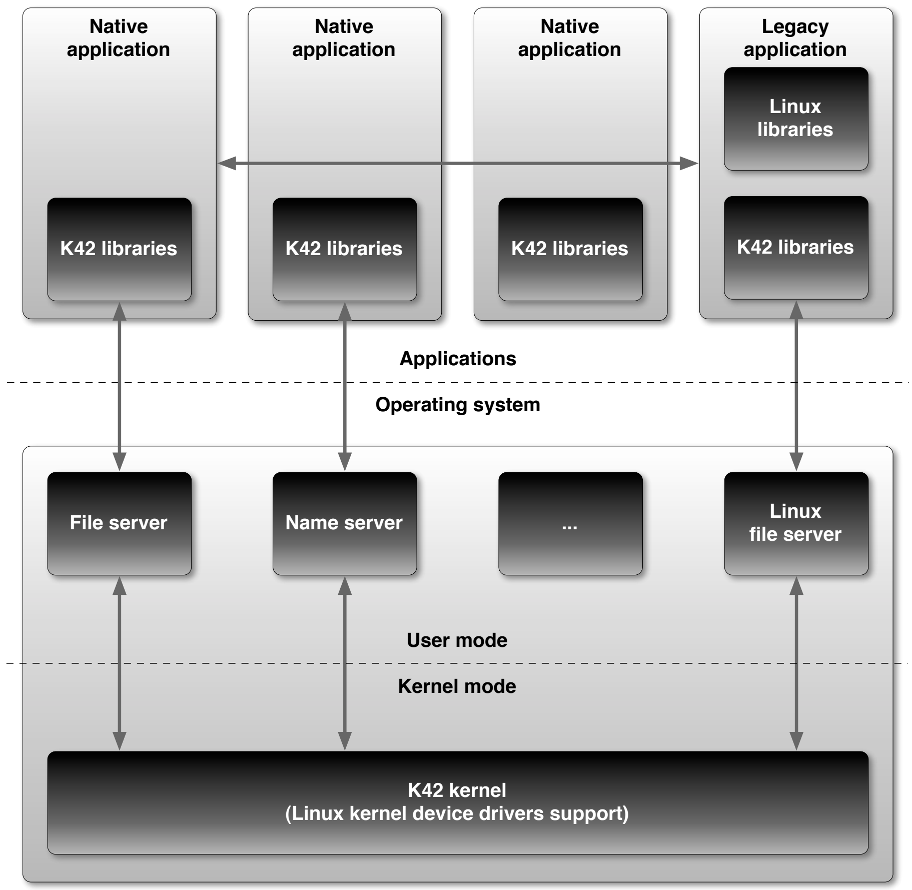


Figure 2.2: K42 operating system structure

As Figure 2.2 shows, K42 supports both the *Linux* Application Binary Interface (ABI)[6] and Advance Programming Interface (API)[7] thanks to an emulation layer running in user mode. This Linux emulation layer implements Linux system calls by method invocations on K42 native libraries. Hence, when an application is developed for K42 it is possible to either use the Linux "personality" or the native and K42 interface.

In addition to Linux system calls a porting of the *GNU is Not UNIX (GNU) C Library (glibc)* dynamically loadable is available to guarantee a really good compatibility. However, programming against the native interface is obviously preferable since it is possible to fully exploit those feature of K42 that makes it so much interesting when we speak of autonomic computing.

The Linux "user mode personality" is not the only "portion" of Linux present in K42, the Linux "kernel mode personality" is available too, this means support for Linux device drivers and modules implementing file systems or network protocols is provided too.

It is important to note that even though Linux is currently the only "personality" supported in addition to the native one, K42 is theoretically capable to implement more then one "personality" [30, 26].

2.3.2 User mode implementation of system facilities

In K42 many key system facilities, which usually resides within the kernel code, thus are executed in kernel mode, are programmed to run in user mode.

[6]Describes the low-level interface between two program (*e.g.*, application and operating system). It covers details such as data type size and alignment, routing calling convention, which controls how input values are passed and output values are retrieved, and the binary format of object files and libraries.

[7]Set of routines, data structures, object classes and protocols provided by libraries, services or operating systems in order to support application building.

All *thread scheduling* has been moved to user mode. The kernel is aware only of user processes and for each user process maintains an entity called dispatcher which is in charge of handling all thread scheduling in user mode. This way switching between different threads belonging to the same process is way more efficient resulting in less context switches between user mode and kernel mode [31, 26]. The same approach has been adopted for *timer interrupts* which are handled by the same dispatcher associated to each process, this is a major improvement since most of the times timers are cancelled before they expire thus avoiding context switches to instruct the kernel on what to do.

Another interesting characteristic is the *page-fault handling* process which is managed by the kernel if and only if the page-fault is caused by kernel code otherwise it is managed by the same dispatcher controlling threads and timers [32, 26]. The same can be said for more general *I/O operation* which are efficiently managed by an event-driven model instead of a more classical polling model, which is obviously less efficient.

2.3.3 Object orientation

As said before, in K42 each resource is managed by a different set of object instances and due to the object-oriented design adopted each object encapsulates the data structures necessary to manage the resource as well as the locking mechanisms to securely access those data structures. This approach tends to reduce the usage of both global data structures and thus global locks on shared data structures.

The object-oriented model adopted in K42 is an enhanced model based on Clustered Objects (COs). Most of the requests an operating system kernel receives usually involve the usage of a single resource while only a little amount of requests involves the usage of more than one resource and thus the necessity to share data. In general, the caller needs to now this details, however, COs are designed in a way the caller is not meant to be aware if its request is independent[8] or dependent[9] thus making the call transparent. COs are in charge to use the best underlying implementation, which can be distributed or shared, to maximize performance and scalability.

Performance maximization and scalability come from the adoption of object-oriented methods; when the amount of shared data structures is reduced, possibly becoming null, no global locks are accessed, thus, different requests to different resources can be processed independently and concurrently; moreover, the object-oriented design guarantees good locality. Per-instance resource management allows multiple policies and mechanisms to be supported simultaneously, and this makes K42 customizable (*e.g.*, two opened files can adopt different caching policies each of each best serves the working condition).

Clustered objects

To provide both uni-processor and multi-processor components within a single object model, the majority of K42's objects are implemented as COs. COs are a possible implementation for partitioned object model supporting both shared and distributed implementations; others implementation of partitioned objects are fragmented objects [24] and distributed shared object [25].

COs are totally transparent to the client, in fact clients see the illusion of working with a single visible object that is actually made up of a set of representative objects each of which handles calls for a different subset of processors. Methods invocations from a processor or processors cluster are usually directed to the representative assigned to that processor or processors cluster; if the representative is the same for every processor the imple-

[8]A request is independent if it uses only resources belonging to the called object.

[9]A request is dependent if it uses resources other than the ones belonging to the called object.

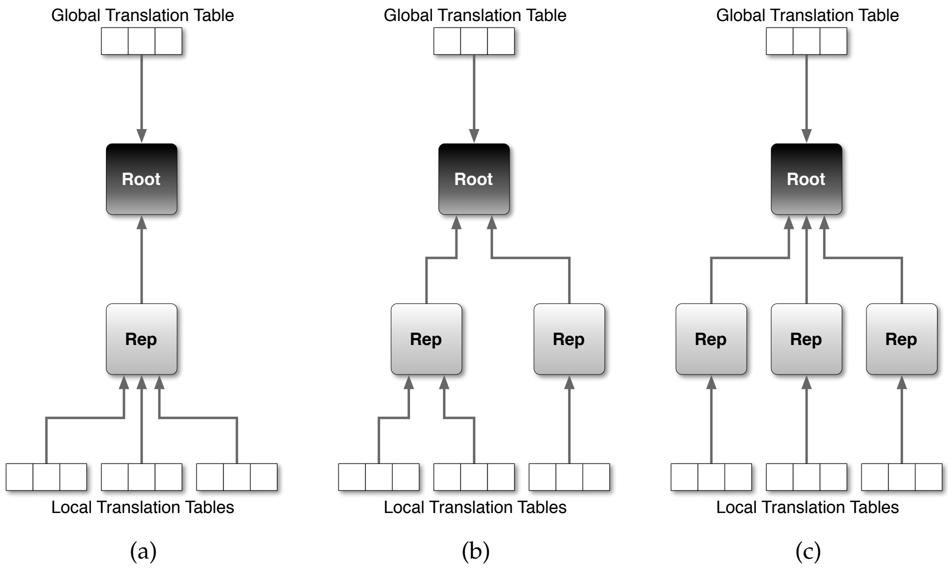


Figure 2.3: CO implementations: (a) shared, the three different LTTs belonging to three different processes point to the same Rep; (b) mixed, two out of three LTTs point to a shared *Rep* while on of the LTTs has a dedicated Rep. The first and the second LTTs form a so called processors cluster; (c) distributed, every LTT points to a dedicated *Rep* achieving a fully distributed solution

mentation is shared (just like it is if you are using standard C++ objects), conversely it is possible that a CO has more representatives, at the extreme one for each processor, achieving a distributed implementation. Figure 2.3a shows a shared implementation, Figure 2.3b shows a partially distributed implementation, while Figure 2.3c shows a fully distributed implementation.

COs help exploiting concurrency fully utilizing multi-processor systems, reducing cache misses and remote memory accesses avoiding - where possible - shared data and false sharing[10] achieving the three main characteristics described in [33, 34, 27].

[10]False sharing is the accessing of different data elements that happen to reside on the same cache line.

K42 is based on COs instead of "ordinary objects" because COs provide a way for implementing multi-processor optimizations behind a fixed interface. COs share three features with common objects:

- a unique interface;
- a single reference per-instance;
- an hidden internal structure.

Moreover, COs have some peculiarity, such as:

- representatives which are hidden to the client and cooperate to replicate, partition, and migrate the data aiming for higher possible locality;
- clients access COs transparently, call are then dispatched to the correct representative associated with a certain processor or processors cluster.

A CO implementation consists of two portions: a *Root* definition providing a shared point to access the CO and a *Rep* definition which provides the per-processor or per-processors cluster portion of the CO. In K42, the base class each *Root* inherit from is called `CObjRoot` which is further specialized in `CObjRootSingleRep` and `CObjRootMultiRep` while the base class each *Rep* inherit from is called `CObjRep`.

Clustered Object System One of the key components of K42 is a module called Clustered Objects System (COS) which is identified as a set of `COSMgr` objects. The COS itself is a CO and there is a `COSMgr` instantiation for each address space providing construction and destruction services. The COS is partitioned in two portion, the user mode portion and the kernel mode portion respectively known as `COSMgrObject` and `COSMgrObjectKernObject`.

The COS provides a variety of facilities:

- for globally identifying a COs. This facility is called Clustered Object Identifier (COID) or Clustered Object Reference (COR), is unique across the system and is used by the clients to access the newly created instance;
- for associating a *Rep* of a CO to a processor or processors cluster;
- for mapping a COID to the appropriate *Rep* given a specific processor or processors cluster;
- for automating the safe reclamation of resources associated with a CO. This is done by means of a semi-automatic garbage collector;
- for allocating resources locally to a specific processor or processors cluster;
- for sharing and communicating between *Reps* or COs. This is provided thanks to the memory management and the thread schedulers/base scheduler.

The instantiation of a *Root* makes the COS assign a unique COID that is used by clients to access the newly created instance. To the client, a COID appears to be a pointer to an instance of a Rep, which is actually accessed within indirections the client is not aware of. To provide better code isolation this mechanisms is performed by means of the `DREF` macro. For example, a client call to a method `foo()` exposed by a CO having its COID stored in `id` would look like: `DREF(id)->foo()`.

A set of tables and protocols are used to translate calls on a COID in order to achieve the unique run time features of COs. There is a single shared table of *Roots* called the Global Translation Table (GTT) and a set of *Reps* tables called LTT, one per processor[11].

[11] The virtual memory map for each processor is set up so that a LTT appears at the same

Construction In order to achieve the lazy creation of the *Reps* of a CO, hence avoiding excessive resource usage and initialization costs, *Reps* are not created or installed into the LTT when the CO is instantiated. Instead, empty entries of the LTT are initialized to refer to a special hand-crafted object called the *default object* identified as `COSDefaultObject` as shown in Figure 2.4a. Therefore, the first time a CO is accessed on a processor or processors cluster (or an attempt is make to access a non-existent CO), the same global default object is invoked. The default object leverages the fact that every call to a CO goes through a virtual function table. The default object overloads the method pointers in its virtual function table to point at a single trampoline method. The trampoline code saves the current register state on the stack, looks up the *Root* installed in the GTT entry corresponding to the COID that was accessed, and invokes a well-known method that all *Roots* must implement called `handleMiss()`. `handleMiss()` is responsible for installing a *Rep* on the processor or processors cluster LTT, see Figure 2.4b. This is done either by instantiating a new *Rep* or by identifying a preexisting *Rep* and storing its address into the address pointed to by the COID in the LTT. On return, the trampoline code restarts the call on the correct method of the newly installed Rep as shown in Figure 2.4c. The above process is called a *miss* and its resolution *miss-handling*[12].

Considering this allocation policy, it is the *Root* that decide how *Rep* will be assigned, hence achieving a shared implementation or a distributed implementation, returning to the default object a pointer to an existing *Rep* or a pointer to newly instantiated Rep.

---

virtual address on each processor but is backed by different physical pages. This allows the entries of the LTT, which are at the same virtual address on each processor, to have different values on each processor. Hence, the entries of the LTT are per-processor despite only occupying a single range of fixed addresses

[12]The miss-handling process is implemented by every *Root* thanks to the set of attributes and methods `CObjRoot` inherits from `COSMissHandler` and `COSMissHandlerBase`

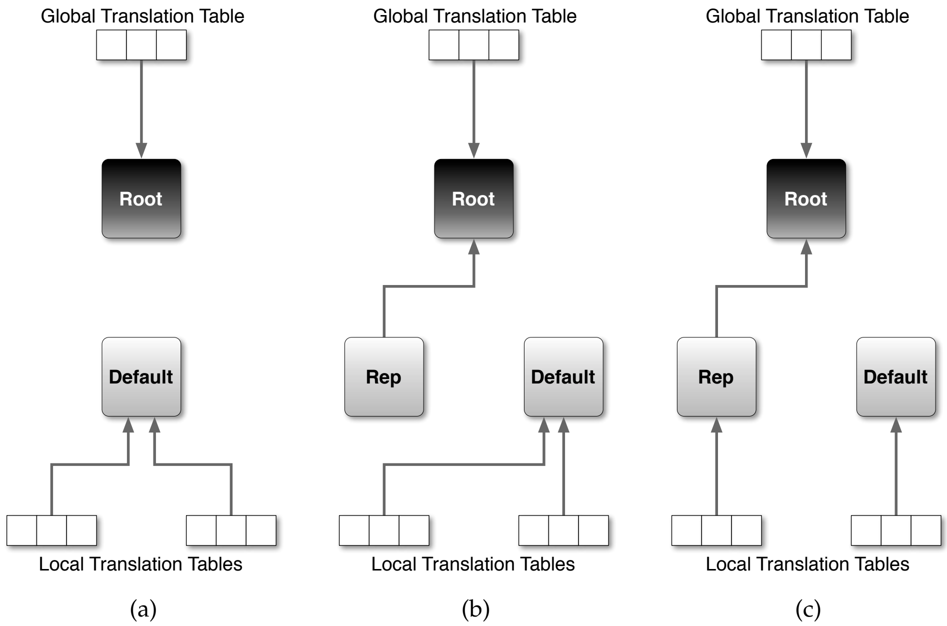


Figure 2.4: Miss-Handling process: (a) the *Root* of the requested CO is installed in the GTT and each LTT points to the default object; (b) a thread calls the CO guided by Root, the call is intercepted by the default object which starts the miss-handling, it instructs the *Root* to provide a *Rep* - the *Root* decides if a new *Rep* is needed or a pre-existing *Rep* is going to be use according to its shared or distributed implementation; (c) the default object installs the *Rep* reference in the LTT of the process the calling thread belong to and restarts the call

Destruction A CO must also de-allocate itself guaranteeing the safe reclamation of its resources and due to the inner nature of COs this is could become a non-trivial task. K42 provide a semi-automatic garbage collection mechanisms through the COS that helps addressing this issue. The garbage collection is semi-automatic because the programmer is in charge to inform the COS about the possibility to destroy a CO. The COS ensures *Reps* are destroyed only after all threads that may potentially have references to the CO have terminated. Further, it only reclaims the *Root* and the COID when processor wide it can be ensured all threads which might have a reference to the CO has terminated[13].

2.3.4 Hot-swap mechanism

For many operating system resources optimize the access for the common pattern is pretty simple and can be implemented efficiently. Although this generic implementation can be valid for most of the time, it sometimes becomes expensive when the resource is accessed with a less common pattern. Hot-swapping allows the resource to be always managed in the best possible way changing the way the resource is accessed.

To give an example, caching is a very common argument and lots of policies are available, each one with its own strengths and drawbacks. Hot-swapping can exploit these policies by monitoring the access pattern to the cache and then changing the policy accordingly.

Another example of this is file sharing. While most applications have exclusive access to their files, on occasion files are shared among a set of applications. When a file is accessed exclusively by one application, an object in the application's address space handles the file control structures, allowing it to take advantage of mapped file I/O, thereby achieving per-

[13]The ability to know when a CO is reclaimable is the consequence of the thread life time approximation implemented in the K42 scheduler.

formance benefits. When the file becomes shared, a new object dynamically replaces the old object. This new object communicates with the file system to maintain the control information for every application.

The *hot-swap* mechanism is powerful and enables the switch among available implementation of the same "living" component. Therefore, an operating system that support hot-swapping is allowed to be virtually optimized for every scenario thanks to its ability to switch between different implementations of the same functionality while it is running. In K42, the hot-swap mechanism follows the general framework proposed at the beginning of Section 2.3 and is completely based on COs and their peculiarity [34].

To implement the online reconfiguration algorithm outlined above, a specialized CO called the *mediator object*, implemented in a distributed fashion, is used. The mediator object - mediator from now on - establishes a worker thread and a *Rep* on each processor for which the target CO - old object from now on - was accessed. The mediator is interposed in front of the old object intercepting all calls for the duration of the switching process.

The mediator transits through a sequence of phases in order to coordinate the switch between the old object and the new object. The mediator's worker threads and *Reps* (remember the mediator is implemented in a distributed fashion) function independently, only synchronizing when necessary in order to accomplish the switch.

Figure 2.5a shows the prior phase in which the old object is invoked normally. Figure 2.5b shows the next phase called forward phase. The first step of the switching process is to instantiate the new object with no assign COID, moreover, the installation of its *Root* into the GTT is skipped. Then mediator is created and passed both the COID of the old object and a pointer to the *Root* of the new object. The mediator then proceeds to interpose itself.

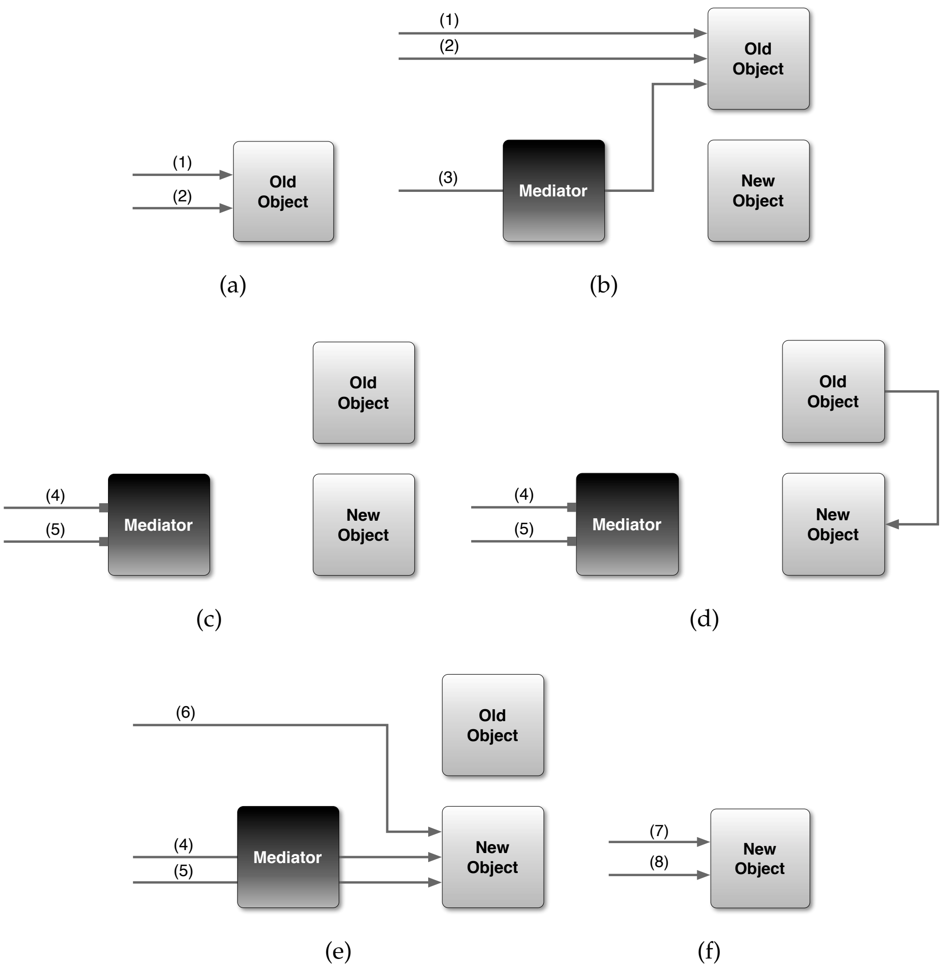


Figure 2.5: Switching process. This Figure shows the phases of the switching process with respect to a single processor: (a) prior phase, the "old object" is in its normal working condition; (b) forward phase, a mediator is interposed and starts to forward calls to the old object, the new object is instantiated; (c) block phase, the mediator stops forwarding calls and forces a quiescent state for the old object; (d) transfer phase, the internal state of the old object is exported, translated, and imported to the new object. All the references to the old object are changed to references to the new object; (e) forward phase, the mediator forwards previously blocked calls and the new object starts receiving calls, the mediator finally de-instantiates the old object and itself; (f) post phase, the "new object" is in its normal working condition

Interposing a mediator in front of the old object ensures that future calls temporarily go through the mediator. To accomplish this, the mediator must override both the GTT entry's *Root* pointer and all the active LTT entries' *Rep* pointers. The GTT entry needs to be changed to handle future misses. To change the GTT the mediator must ensure that no misses to the old object are in progress[14].

The LTT entries must be changed to handle calls to already established old object's Reps. The mediator spawns a worker thread on all the processors that have accessed the old object[15]. These threads have a number of responsibilities, but their first action is to reset the LTT entry on each processor back to the default object. This ensures that future accesses will be directed to the mediator due to the miss-handling process. Therefore, on each new miss the mediator's *Root* installs a new mediator's *Rep* into the LTT accomplishing the interposition.

Once the mediator's *Rep* is installed into the LTT entry, calls that would normally call one of the methods in the old object end up calling the corresponding method in the mediator. The actions that the mediator's *Rep* has to take on calls during the various phases of switching include: forwarding the call to the old or new object depending on the forward phase and keeping a count of active calls, selectively blocking calls, and releasing previously blocked calls[16].

The mediator stays in the forward phase, allowing new calls - referred as tracked calls - to proceed to the old object. The forward phase lasts un-

[14]The miss-handling process makes use of a reader-writer lock associated with each GTT entry, on a miss this lock is acquired in read mode. In order to atomically change a GTT entry, the associated reader-writer lock is acquired in write mode, ensuring that no misses are in progress.

[15]The *Root* of each CO maintains the set of processors for which a miss happened, the mediator can query the old object's *Root* to determine for which processors to spawn threads.

[16]To make the forward process transparent to the client the mediator's *Rep* must not alter the stack layout whenever a call is forwarded.

til the mediator determines that a quiescent state can be reached by ending the forward phase. More precisely, the mediator determines that there are no more threads started before the switching process initiation still accessing the old object. To detect this, the worker thread utilizes an Read Copy Update (RCU)-like mechanism based on the *generation count* feature of K42[34]. The forward phase and the transition to the block phase happen independently and in parallel on each processor, hence, no synchronization across processors is required.

In Figure 2.5c the mediator reached the so called block phase. In this phase, each mediator's *Rep* establishes a quiescent state on its processor by blocking all new calls while waiting for any remaining tracked calls to complete. However, because a tracked call currently accessing the old object might itself call a method of the old object again (*e.g.*, recursive calls), care must be taken not to cause a deadlock by blocking any tracked calls. This is achieved by checking the Thread Identifier (TID) determining if the thread is in a tracked call. This also ensures that concurrent switching processes of multiple do not deadlock. If a thread in a tracked call, in one object, calls another object that is in the blocked phase it will be forwarded rather than blocked, thus avoiding the potential for inter-object deadlocks. Finally, the worker threads must synchronize prior to proceed to ensure a "global" quiescent state across all processors.

Once the blocked phase completed, the Transfer phase begins. Figure 2.5d shows this phase in which the worker threads are used to export, translate, and import the state of the old object into the new object. To assist this phase a transfer negotiation protocol is provided. For each set of functionally compatible COs, there must be a set of state transfer protocols COs comply with.

The worker threads synchronize again to allow the safe replacement of the GTT entry currently pointing to the mediator's *Root* with the reference to the new object's Root. All the worker threads then cease call interception by modifying the LTT entries to point to the new object's Reps. The worker threads then resume all threads that were suspended during the block phase and forward them to the new object as Figure 2.5e shows. The worker threads deallocate the old object, the mediator's *Reps* and then terminate. Figure 2.5f shows the post phase in which the new object is invoked normally.

2.4 Summary

As presented in this Chapter, a common and "somehow" straightforward approach to implement self-aware adaptive computing systems to handle varying environments implies the use of adaptive code based software. However, adaptive code does not achieve the full autonomic vision previously outlined because of three main disadvantages: requires foreknowledge, produces higher code complexity, and causes higher overhead [30]. Requiring foreknowledge means for an application to be thought, designed, and implemented bearing in mind every possible scenario the application may run into. Given that application development is an error prone task, this additional constraint tends to produce high complexity code since every situation needs to be hard-coded. In the end, the more the code is complex the more it introduces overheads resulting in an adaptable application with lower performance in every scenario it covers with respects to a set of non-self-aware adaptive applications each of each is thought, designed, and implemented to fit a unique scenario.

A more fundamental limitation of the adaptive code solution is its unsuitability for the larger vision of self-aware adaptive computing we outlined in Chapter 1. Adding new scenarios and functionalities to an adaptive code based application means introducing a considerable amount of new code. This fact highlights the inefficiency in terms of maintainability of this solution.

A novel and promising approach in self-aware adaptive computing systems is the adoption of online reconfiguration techniques. On a case-by-case basis, the infrastructure allowing the software to change on-demand is a lot more complicated than the adaptive code alternative, however, this infrastructure needs to be implemented only once since it is reusable for every application. Given the implementation of the online reconfiguration infrastructure we obtain lower code complexity since every piece of software needs to address a specific situation therefore the overall solution is better in terms of maintainability and ease of development.

Coming back to the solutions presented in Chapter 2, the adaptive algorithm selection framework built upon STAPL represents a notable step forward to the common adaptive code approach. Both the monitor and the decision code are decoupled from the portion of code actually implementing the desired functionality. This is a major improvement with respect to the common "monolithic" approach adaptive code usually imposes.

The major shortcoming we see in this framework is the requirement to "work" just before the application starts running, at launch time, this prevent any kind of online adaptation failing to fulfill one of the most relevant autonomic property. This is due to the decision making code which is based on "offline" techniques which are not usually suited to operate during the run time of applications.

With respect to the adaptive algorithm selection framework presented, the approach adopted by the K42 operating system is way more ambitious. Providing the online reconfiguration support directly inside the operating system is, in our opinion, the best solution to reach a considerable amount of applications with the benefit of unifying the overall interface thus the way applications make use of the online reconfiguration capability.

All the operating systems presented to enable online reconfiguration in reconfigurable computing systems suffer from the same issue: on one hand hardware/software co-design techniques are useful to exploit reconfiguration capabilities, on the other hand, the decision of what functionalities are executed in hardware and what functionalities are executed in software is bounded at design time reducing the freedom of the reconfigurable computing system and the ability to adapt the execution upon the varying internal and environmental conditions.

The K42 operating system provides a great sub-system to adapt its behavior according to every scenario and this novelty is the result of a whole new implementation. Starting from scratch surely granted the possibility to design the whole operating system with adaptive concepts in mind, however, implementing a whole new operating system means its features, even though they are unmatched, are not going to be widespread for a potentially long time period considering the low reaction time the market is accustomed to when it comes to operating systems. Moreover, the K42 operating system does not provide applications with a unified sub-system to monitor performance and a unified sub-system for decision making being very tailored on adaptability. Being so much focused on adaptability, the K42 operating system can be classified as an adaptive solution instead as a self-aware adaptive solution.

Chapter 3 will introduce the proposed approach to implement self-aware adaptive computing systems underlying the advantages with respect to the *state-of-the-art* solutions presented in this Chapter.

Chapter 3

Proposed methodologies

In Chapter 2 we overviewed some of the most interesting solutions embracing the autonomic vision proposed in Chapter 1. However, as underlined at the end of the previous Chapter these solutions implement self-aware adaptive capabilities without fulfilling the overall autonomic picture presented.

The development of self-aware adaptive computing systems is strictly linked with the basic capabilities the Observe, Decide, Act (ODA) loop provides: observation, decision, and action where action means not only execution but also adaptation of the computing system. We think a self-aware adaptive computing system cannot prescind from any of these capabilities.

Aim of this Chapter is to propose two distinct methodologies for the development of self-aware adaptive computing systems based on the two approaches presented, adaptive code and online reconfiguration. Our former methodology is based on the adaptive code. Nonetheless, we judge online reconfiguration a far better approach with respect to adaptive code, thus our adaptive code solution is strongly biased toward online reconfiguration adopting some of its key points where necessary. The latter methodology is purely based on the online reconfiguration approach and is inspired to the K42 operating system while taking some of the concepts and functionalities introduced in operating systems supporting reconfigurable hardware devices. Even though injecting self-aware adaptive capabilities in the operating system is the road to follow we are not willing to develop a whole new operating system from scratch since the amount of time needed to develop a fully mature one is tremendous with respect to the time needed to create specific self-aware adaptive applications. Although the design from scratch is the strong point of the K42 operating system it is also its main weak spot as highlighted at the end of Chapter 2. To reach a wide set of applications our online reconfigurable solution is based on a well-known and widespread solution: the *Linux kernel* which proves to be a good platform for injecting self-aware adaptive capabilities as it was for supporting reconfigurable hardware devices.

The remainder of this Chapter is organized as follows: Section 3.1 proposes and defines the concept of self-aware adaptive process providing a useful abstraction for self-aware adaptive computing systems. The following Sections 3.2, 3.3, and 3.4 describe how the proposed methodologies solved the three issue of observing, deciding, and acting meanwhile fulfilling the autonomic scenario described.

3.1 Self-aware adaptive process definition

The first step toward the design of a self-aware adaptive computing system is the definition of the entities running on the computing system. On one hand, the computing system must provide support for common (*i.e.,* non-self-aware adaptive) applications, on the other hand, the computing system must provide to the applications those enabling technologies that allow the exposure of a self-aware adaptive behavior. For example, different implementations of the same functionality should be available and the best[1] one is autonomically chosen and used according to the internal state and the environmental conditions.

Redefining, or better, extending the concept of process, the main actor in computing systems, to improve the generality of the proposed methodologies and to fit the autonomic scenario is thus necessary. The concept of process is extended becoming a *self-aware adaptive process* providing a uniform representation of both common and self-aware adaptive applications. The concept of self-aware adaptive process models the adaptive usage of multiple resources (*e.g.*, both hardware implementations and software implementations of the same functionality) and the seamless switch among them that happen while the process is executing. Informally, a self-aware adaptive process can be viewed as an abstraction of a concrete process that is fasten to general purpose operating systems.

Since a self-aware adaptive process is in abstraction of a concrete process in terms of the states it can enter, we need to define the process state diagram and we are taking the Linux kernel as an example. A process can be in one of the following states [35]:

- *ready*: the process is waiting its quantum in order to be executed on a processor;
- *execute*: the process is being executed on a processor;
- *suspend*: the process is sleeping until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of condition that might wake up the process;

[1] Best does not necessarily mean the implementation that grant the best performance in terms of run time. The choice of the best implementation is bound to the definition of performance, for example, consume the least amount of power, maximize the overall throughput, and so on.

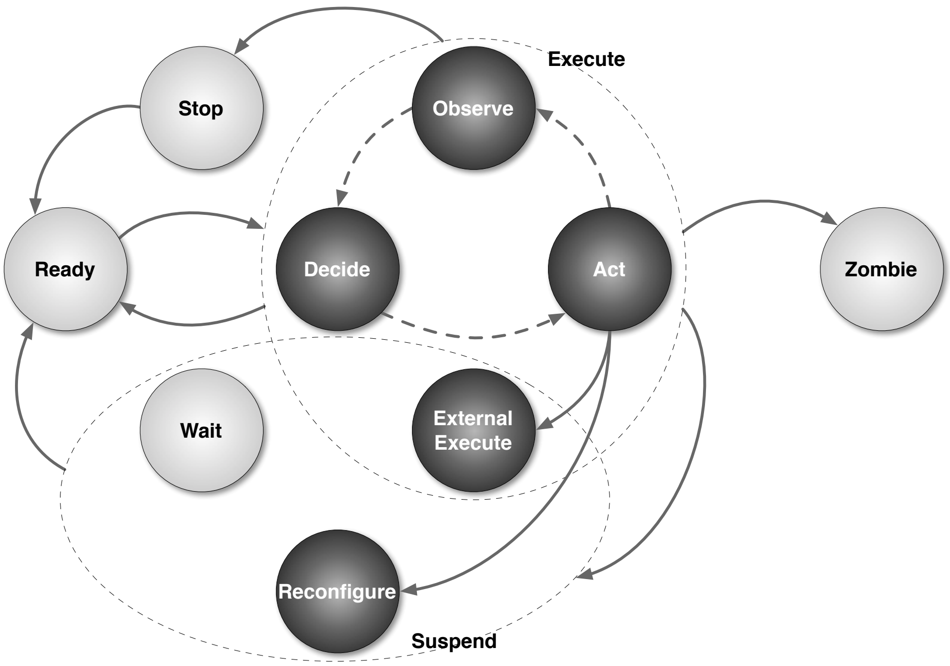


Figure 3.1: Self-aware adaptive process state diagram

- *stop*: the process is being prevented to execute on a processor due to an external intervention and only an external intervention can change the process state (*e.g.*, signal like `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, and `SIGTTOU` can induce this process state);
- *zombie*: the process is being terminated due to execution ending but the parent did not issue a system call from the `wait()` family to return information about the dead process. The kernel is prevented to discard the data connected to the dead process until the parent dismisses it.

A self-aware adaptive process is mapped on the very same set of process states reported above. However, to fully understand all the events that can happen and all the activities executed a more abstracted process state diagram must be introduced. This process state diagram: the *self-aware* *adaptive process state diagram* is shown in Figure 3.1. There are two states that need to be further described with respect to the previous list:

- *execute*, the self-aware adaptive process is being executed on a processor. The execution can happen either on a general purpose processor and in this case the state can be *observe*, *decide*, *act*, where act means both execution and adaptation, or it can happen on a co-processor (*e.g.*, General Purpose Graphics Processing Unit (GPGPU), Field Programmable Gate Array (FPGA), and so on) and in this case the state is *external execute*;
- *suspend*, the self-aware adaptive process is either executing on a co-processor hence its state is *external execute* or sleeping until some condition becomes true therefore its state can be either *wait* or *reconfigure*.

The state *external execute* is “shared” between the “macro state” *execute* and the “macro state” *suspend* indicated by means of dashed ovals in Figure 3.1. On one hand, from an abstract point of view, when a self-aware adaptive process is in the state of external execute it is actually using processing resources then its “macro state” should be execute. On the other hand, from a concrete point of view, when a self-aware adaptive process is in the external execute state it has just performed a series of Input/Output (I/O) operations (*e.g.*, memory I/O and bus I/O to pilot a peripheral or a co-processor) hence its “macro state” is suspend since it must be awakened from the arrival of the desired result before it can continue its execution.

The state *reconfigure* is another abstraction with respect to the “macro state” *suspend*. When a self-aware adaptive process is running on a computing system equipped with reconfigurable hardware devices it can require a reconfiguration in order to speed-up its execution thanks to the mapping of co-processors on the reconfigurable hardware device’s are. In concrete, an operating system supporting reconfigurable hardware devices sees this reconfiguration process just like another set of I/O operations. However, a reconfiguration can represent a really important kind of I/O operation for a self-aware adaptive process hence there is an ad-hoc process state to describe this situation.

A deeper analysis on what happens in those states a self-aware adaptive process can enter is discussed in the reminder of this Chapter.

3.2 Observe

As presented in [36], a self-aware adaptive computing system must be able to monitor its behavior to update one or more of its components (*e.g.,* hardware architecture, operating system, and applications), in order to achieve its goals.

A self-aware adaptive computing system needs a lightweight and "real time" monitoring framework. The *Application Heartbeats* framework - Heartbeats from now on - described in [36] implements a simple yet extremely effective and powerful monitoring infrastructure. More precisely, it is a library made of a really small set of functions that makes it straightforward to use.

Heartbeats makes it possible to declare performance goals by means a simple concept, the *heart rate*. The advantage of such a simple concept is the fact that it can be translated in a straightforward way into a common performance measures such frames per second (*i.e.,* for encoding, gaming, and rendering applications) and computations per second (*i.e.,* for cryptographic applications where each computation is identified by a set of blocks).

Defining a goal means defining an extremely important parameter for modern computing system: the *Quality-of-Service (QoS)*. In these days where the computational power of a computing system is skyrocketing and "comes-for-free" the "as-fast-as-possible execution" is loosing its predominance in favor of measures such as the overall throughput or, for particular environment, the power consumption. The concept of QoS sums up in a good way these performance measures. The QoS is determined within an heart rate window between the minimum heart rate and the maximum heart rate. Other important parameters to finely control every possible aspect of the monitoring and hence the QoS are the window size, the history size, and so on.

Even though defining a performance goal is fundamental, this is only the first step in using Heartbeats. During its execution an application registered with the monitoring framework must provide information about the progress of the execution allowing Heartbeats to update the data associated with the application behavior. An application updates its progress through a series of calls that signify a series of *heartbeats*. The series of heartbeats in a time window enables the computation of the overall heart rate. The library automatically updates all the necessary information about the heart rate and such information is made available to a designated observer which can be either the application itself or an external entity to trigger the decision process.

Figure 3.2 presents the general structure of the resulting self-aware adaptive computing system with the introduction of Heartbeats as an available service. In this picture we notice both the two scenarios previously mentioned. In the former scenario, *Application 0* is a standalone application featuring its own observation process therefore the execution is constrained by its own logic. In the latter scenario, *Application 1* does not feature the observation process which is implemented separately in *Application 2* that reads information about *Application 1* execution by means of Heartbeats library and communicates with *Application 1* eventually constraining its execution.

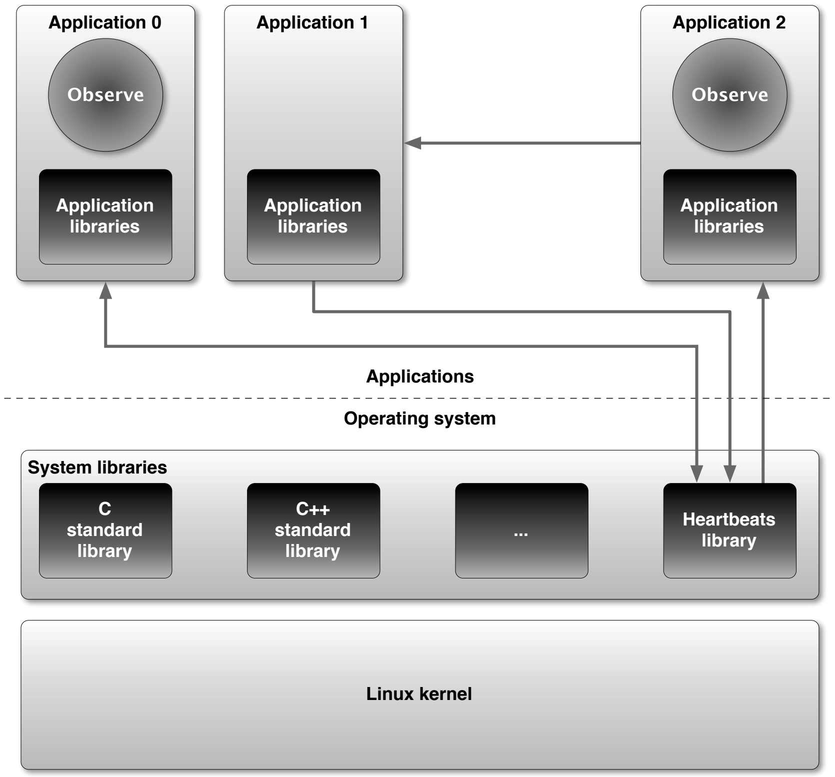


Figure 3.2: Self-aware adaptive computing system with monitoring

3.3 Decide

For a self-aware adaptive application the ability to "think" and decide in order to fulfill its performance goals is complementary to the ability to observe its behavior, therefore, they are equally important. The decisioning framework is fed with data collected by the monitoring framework hence they are strictly tied up. In the remainder of this Section we will see how the choice of the monitoring framework influenced the choice of the decisioning framework.

As a monitoring framework we chose a lightweight solution: Heartbeats. Heartbeats represents a simple yet powerful library which is suitable to work while the system is running and can provide a useful approximation of the throughput of a self-aware adaptive application.

Quantifying the effects due to the environmental and the run time conditions on the internal state of a self-aware adaptive computing system is a task that can be accomplished in several different ways. In particular, we can make a first distinction between two broad categories of models needed to take decisions: *analytical models* and *empirical models*. The former family is composed by manually generated models that implicitly contain both architecture and implementation related information in their formulation. Though usually requiring more effort from the developer, analytical models can take advantage of the provided knowledge which often results in increased accuracy. However, these models tends to be "static" since it is nearly impossible to abstract the changing number of variables that usually affects the execution of an application. The latter family of models uses information collected during the execution of applications to generate a suitable representation to decide which implementations and parameters must be adopted for a certain execution. They are desirable because they implicitly make use of both environmental and run time conditions; however, they are surely an approximation of the reality, therefore generating a suitable model can sometimes be difficult.

Even though the usage of analytical models can help achieving an accurate representation of the reality, such technique is not always applicable to self-aware adaptive applications that need a fast decisioning process to enable run time adaptation. Moreover, the use of analytical models does not fit the monitoring framework we selected.

The broad class of empirical models is more suited to cope with the monitoring framework we adopted. Considering the highly dynamic nature of the environment self-aware adaptive computing systems are meant to work in, the adoption of "offline" decisioning solutions that need to construct a basic model before being used, the so called eager learners (*e.g.*, decision trees), is not feasible.

Since the focus of this thesis is the design and development of an hot-swap mechanism for adaptive systems we did not spend a lot of time searching and implementing a *state-of-the-art* decision process, we came up with an agile (*i.e.*, lightweight, low-overhead) and simple decisioning process that belong to the broad family of empiric models based on heuristics. Different works regarding self-aware adaptive computing systems developed by our research group are more focused on decision processes and adopt techniques varying from machine learning [37] to control theory [38].

Even though heuristics tend to generate representations of the reality even more approximated than models generated by machine learning techniques coupled with offline decision techniques (machine learning coupled with offline decision is consider an empirical way to take decision), a simple yet effective heuristic can produce a viable approximation. The adoption of heuristics which limit the overhead on the self-aware adaptive computing system allowed us to enable periodic performance check and hence periodic run time adaptations to better fit the requested QoS. The need for run time adaptation is given by the fact that the system is *live* and *lives* in an unpredictable environment: for such reason it is impossible to decide statically which way of working proves to be the most convenient. For instance a static analysis might show that for any given input data size there exists an implementation which outperforms the others; yet many other factors, such as the variable system work load, might prevent the static analysis to get real. Since a self-aware adaptive computing system is approximate, the system must monitor its internals and the surrounding context and then take decisions accordingly, dynamically balancing all the constraints, without searching for the optimal solution at all costs.

Since this thesis work is focused on self-aware adaptive computing systems featuring hot-swap mechanisms to adjust the execution and nonetheless to fulfill their goals, the heuristic at the base of the decisioning framework must choose which implementation of the required functionality is better for the current run time and environmental conditions and eventually trigger the switch between the active implementation with another one available during the run time. Heartbeats makes possible for the decisioning framework to query the development of the execution and obtain an overall history of the heart rate. Fed with such information, the decision mechanism chooses at run time the best implementation to use in accordance to given constraints (*i.e.,* desired heart rate). The heuristic at the base of the decision mechanisms is thought to void excessive oscillations among the available implementations; this situation may occur if the computing system, which is trying to fulfill the given goal at all costs, is suffering from excessive load rendering every available implementation inadequate to reach the goal.

3.4 Act

The amount of actions a self-aware adaptive computing system can take to adjust its behavior and hence performance to fit the required QoS is potentially enormous and depends on the amount of variables (both internal and external) influencing the QoS which is our performance representation. For example, if the performance goal is: minimize the power the computing system is dissipating, a possible set of valid countermeasures could be: reducing the voltage of cores, switching on/off a certain number of cores, reducing the clock frequency of cores, and so on. This list can be further extended with a set of actions depending on the underling hardware architecture capabilities (*e.g.,* reducing the voltage or the clock frequency of a single core or of a group of cores instead of the whole processor, such a feature is available in some of the latest Intel and Advanced Micro Devices (AMD) processors). Either way, if the performance goal is more "traditional" such as: encode a video at 25 frames per second, a set of valid actions may be: augmenting the number of core the application can use, changing the underling algorithm and the associated data structure, and making use of hardware accelerators instead of software libraries.

The main focus of this thesis is to enable runtime adaptation through hot-swapping which means introducing the ability of switching among different implementations of the same functionality, each of each is meant to be optimal for certain internal and environmental conditions. The hot-swap mechanism should be as general as possible allowing to switch from software to software, from software to hardware, from hardware to software, and from hardware to hardware.

We think that for a self-aware adaptive computing system the ability to switch among different implementations of the same functionality while the system is running is fundamental. However, to hot-swap an implementation with another one is a non-trivial process; different threads within a single process can access data structures concurrently and different implementations can use completely different data structures. State quiescence and state translation are the most visible problems the hot-swap mechanism should look forward to when it comes to hot-swap one in favor of another one. In the adaptive computing literature this is a well-known problem and a general framework to solve it has already been proposed in [23]. This *state-of-the-art* framework inspired both the paths we are pursuing.

In [39] a set of requirement for implementing dynamic update in an operating system is reported, some of them may be ported in the hot-swap mechanisms context resulting in the requirements reported below.

- *Switchable unit*, depending on the implementation of the computing system, a unit may consist of a function, of a data structure, of an object, of a library, or of an entire service or middleware layer. The structure of the system dictates what is a feasible choice. In every case, there must be a clearly defined interface to the unit. Furthermore, external code should invoke the unit in a well-defined manner, and should not arbitrarily access code or data of that unit.

- *State quiescence*, switches should not occur while any affected code or data is being accessed. Doing so could cause undefined behavior. It is therefore important to determine when a switch may safely be executed, when a switch may safely occur the switchable unit is said to be in a quiescent state.

- *State translation*, when a switch is performed affecting data structures, or when a switchable unit maintains its internal state, the state must be transferred, so that the switched unit can continue transparently from the unit it replaced. The two switchable unit may not agree on the data structure they use hence a translation mechanism must be provided. A translation may consist on using a “canonical” format, or passing a reference to the data structure and let the switched unit handling the translation, or marshaling and de-marshaling the data structure.

3.4.1 Adaptive code

The adaptive code based approach is used as a user mode solution to provide an hot-swap mechanism. The result of the proposed methodology is a set of libraries, many different "implementation" libraries and an *adaptive* library. The adaptive library is the one in charge to access in a consistent fashion the active implementation. Each of the available implementation library is tailored towards a specific scenario using different computation resources.

Given the proposed methodology to implement a user mode hot-swap mechanism, we identified the switchable units as the implementation libraries. Since each switchable unit should be self-contained we built each implementation library as a standalone object, more precisely each implementation library is a Dynamic-Link Library (DLL) (*i.e.,* Shared Object (SO) on UNIX operating systems) just like the adaptive library. The adaptive library is introduced to prevent arbitrary accesses to the code of the available implementation libraries and in in charge to provide both adaptability and self-awareness.

Since the result of the adaptive code based approach is a set of libraries made up of an adaptive library and many implementation libraries that behaves just like any other "normal" library, the "state" is kept within each function call applications do to the adaptive library. This means concurrent function calls coming from different threads belonging to the same process have their own "state" and, therefore, there is no need to reach a quiescent state since the state is always stable across the execution of a function call.

The advantage of using the adaptive library which acts as a proxy between the caller and the callee is the fact that the caller does not need to be aware of the presence of many different implementation libraries and their specifics such as the underling data structures and eventually also the interface each implementation library exposes. In fact, the adaptive library make use of “canonical” data structures. The translation among the different “canonical” data structures and specific data structures is handled using the “canonical” data structures as an intermediate representation and is done *on-the-fly* when the adaptive library invokes the the function calls exposed by the active implementation library.

The adaptive library is further extended to introduce awareness, obtaining an *aware adaptive library*. It is instrumented to make use of Heartbeats to signal an observer how the active implementation library is performing hence allowing the entity in charge to take decisions if it is the case to hot-swap another implementation library in favor of the active one or maintaining the current situation. An extensive explanation of the methodology proposed for the user mode applications is available in [40]. Furthermore, Figure 3.3 shows a specialized version of Figure 3.2 in which the concept of aware adaptive library is represented.

Our methodology to build aware adaptive libraries and hence aware adaptive applications allows a lot of flexibility. Running on a suitable operating system supporting reconfigurable hardware devices (*e.g.,* the Linux kernel, which is one example of operating system that support reconfiguration) or on a computing system equipped with hardware accelerators, DLLs implemented using software-only solutions can be switched with DLLs making use of this amount of hardware to increase the overall performance of the application. Once again, the aware adaptive library using hardware based implementation libraries makes the execution and, where needed, the reconfiguration process too, completely transparent to applications.

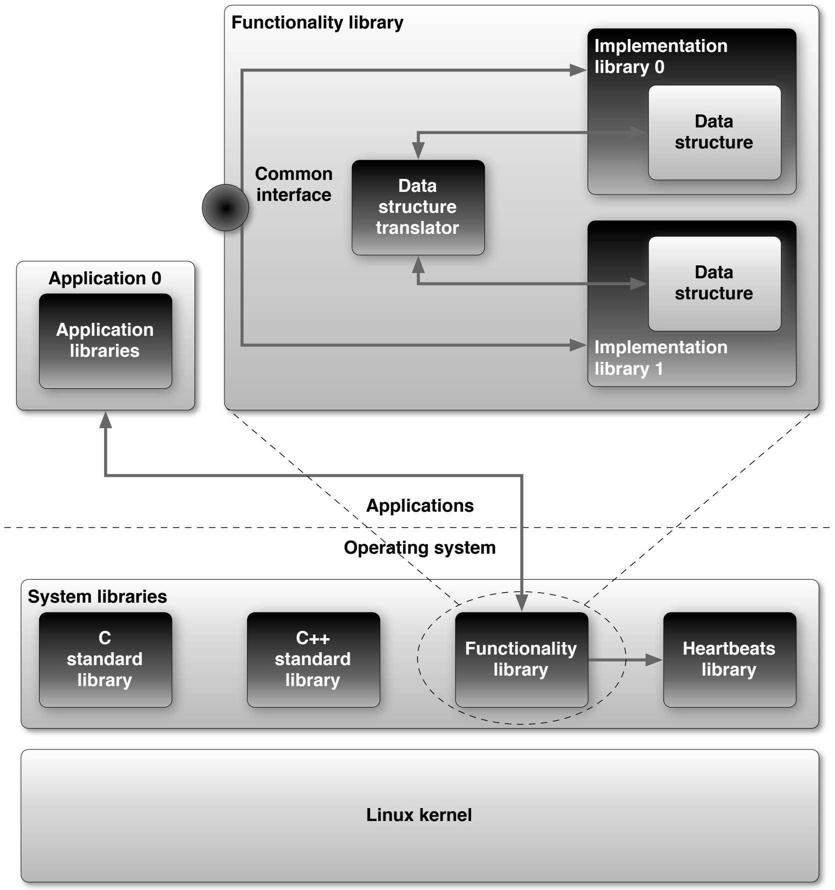


Figure 3.3: Self-aware adaptive computing system with switching

3.4.2 Online reconfiguration

In addition to the *user mode* approach we are also working on a *kernel mode* approach. The kernel mode approach is, at the moment, an enabling technology to provide, in future works, the self-aware adaptive behavior described in Chapter 1.

The K42 operating system allows the online reconfiguration of every object conforming to a certain interface defined in the operating system. This means every object, from user mode objects to kernel mode objects, can be switched while the system is up and running.

In [23] the K42 operating system development team gave few hints about the possibility to implement some of the K42 operating system techniques in the Linux kernel. However, the amount of effort spent on the Linux kernel to provide adaptive capabilities is not as high as it should be to bring the Linux kernel at the same level of the K42 operating system with respect to adaptive capabilities. Therefore, we decided to put a lot of effort on the Linux kernel and followed the road to implement the adaptive capabilities discussed so far in a widespread operating system. Following those hints reported in [23] we aimed for the Loadable kernel modules (LKMs) as the switchable unit that can be hot-swapped within the Linux kernel.

Once the switchable unit (*i.e.,* LKMs) has been identified, the two remaining problems to implement an hot-swap mechanism are: the state quiescence and the state translation. As said before, reaching the state quiescence is a really complex problem, the precondition to solve this problem is being sure no one is using the switchable unit. Within the K42 operating system this problem has been solved using the Object Translation Tables (OTTs) infrastructure, which is used to transparently interface applications and Clustered Objects (COs), and ad-hoc COs, which are interposed between the OTTs and the COs to help reaching a quiescent state. We thought a similar approach for our online reconfiguration approach;

an intermediate component that helps reaching a quiescent state is "interposed" between applications and LKMs, more precisely, the interposition is performed between the system calls layer and LKMs. The state translation problem previously highlighted becomes even more important when it is related to kernel mode data structures; translating the state contained within a LKM into a valid state for another LKM is a non trivial-process since there is no standardized interface to which LKMs must conform when it comes to internal data structures. In addition to this, the Linux kernel lacks facilities to export and import the state of a LKM. Even excluding both the problem reported above to hot-swap a LKM in favor of another one may be a pretty tough process.

Recalling what we learnt about clustered objects in Section 2.3.3, we designed a similar structure to allow the hot-swap of LKMs. The key difference is that COs are structured as outlined due to their ability of being either shared or distributed on-demand while LKMs will adopt this structure solely to allow the implementation of an hot-swap mechanism.

Therefore, a LKM is split in two components, actually two different LKMs, the first one exporting the functionality while the other implementing the functionality. We can see the first acting as a front end and the second acting as a back end of the same system.

The front end LKM, the one in charge to provide the functionality exporting the standard interface the Linux kernel requires, while the back end LKM, the one thought to provide an implementation of the functionality, provides a set of functions conforming to a predefined interface every back end LKM, implementing the same functionality, must comply with.

Figure 3.4 illustrates how the self-aware adaptive computing system is changed due to the introduction of kernel mode switch service. The square containing a LKM split implementation shows how the same front end LKM can be used to access all the implementations of the same functional-

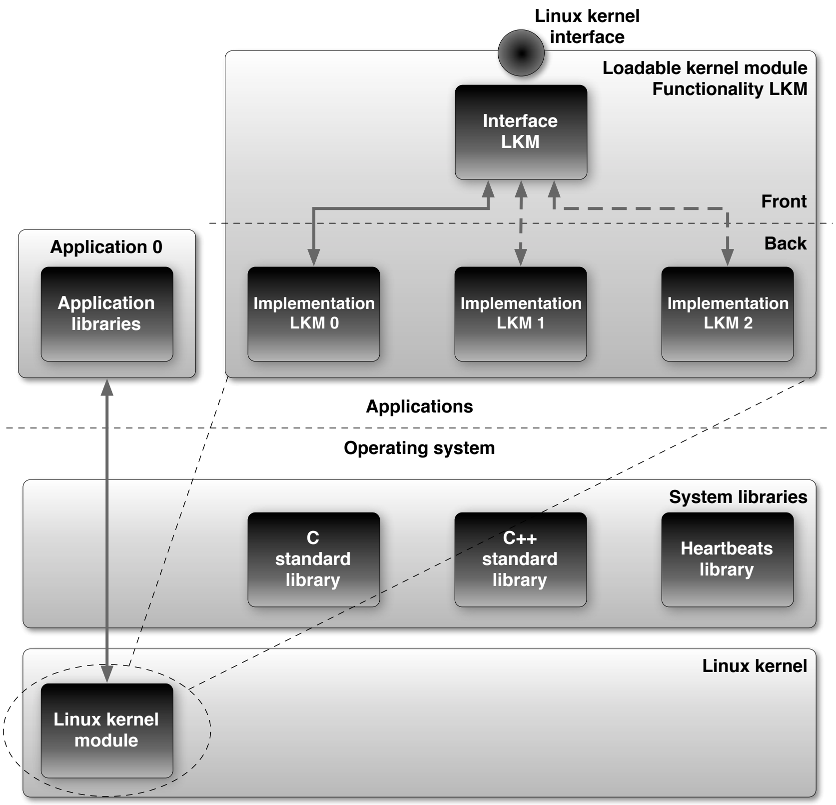


Figure 3.4: Self-aware adaptive computing system with kernel mode switching

ity. Two out of the three implementation available in the example are connected with double-arrowed dashed lines because in a certain time instants only one of the available implementations can be linked to the kernel. This is due to the fact that all the back end LKM must export the same set of functions.

3.5 Summary

We said the approach described in Section 3.4.1 is based on adaptive code while being biased toward online reconfiguration. The proposed methodology is considered more or less biased towards online reconfiguration depending on the fact that implementation libraries are loaded at runtime and not loaded at launch time or are statically linked. In addition to this we can assert our approach is definitely inherent to online reconfiguration since it solves one of the issue regarding the framework for adaptive algorithm selection in Standard Template Adaptive Parallel Library (STAPL), the adaptive code solution we are confronting with. In fact, it allows the switch among the available implementations of the exposed functionality while applications using the aware adaptive libraries are running.

The aware adaptive library defined becomes a self-aware adaptive library when it is coupled with the decision mechanisms described in Section 3.3. The result is a self-aware adaptive solution which monitors the progress of its execution by means of Heartbeats, takes decision upon the measured heart rate and the expected heart rate, and finally adapts its behavior using the hot-swap mechanism.

The online reconfiguration approach described in Section 3.4.2 is ambitious since it proposes to port some of the key functionalities of the K42 operating system inside the Linux kernel which was not designed with adaptive capabilities in mind. This first solution imposes the use of the same data structure for every different implementation, however, adopting the factory design pattern for data structure allocation may help in solving this issue. The kernel mode hot-swap mechanism proposed implements the key features of the hot-swap mechanism of the K42 operating system. The advantage of applying such a technique to the Linux kernel is the possibility of reaching a potentially large amount of installations in a short time period thanks to the wide adoption. At the moment, the only limitation of this approach with respect to the K42 operating system regards the possibility to have either shared fashioned or distributed fashioned of the same LKM running concurrently. However, implementing such a functionality is beyond the scope of this thesis.

Beyond the scope of this thesis is also to provide a lightweight observation solution to enable the monitoring for LKMs. However, the same concept behind Heartbeats can be used in kernel mode. Even though the Heartbeats framework is born as an user mode solution it is portable with minimal effort in kernel mode thanks to its simple approach in terms of both code and concepts. This flexibility is another reason for choosing Heartbeats as our monitoring to enable observation and hence decisioning and adaptation in user mode applications.

In this Chapter we have explained how the self-aware adaptive behavior we are proposing to realize is enabled for user mode applications thanks to the former methodology described. Moreover, an introduction on an enabling technology for introducing the same self-aware adaptive behavior directly inside the operating system kernel we chose, the Linux kernel, has been discussed. Finally, we have highlighted the novelty of our methodologies with respect to *start-of-the-art* solutions presented in Chapter 2. Chapter 4 will illustrate the details about the implementation of the two exposed methodologies.

Chapter 4

Proposed implementation

Chapter 3 presented the proposed methodologies to implement self-aware adaptive computing systems. The former methodology presented is meant for user mode applications and involves the development and the adoption of self-aware adaptive libraries able to switch among different implementations, software ones and, where possible, hardware ones, to fulfill given performance goals. The latter methodology presented is an enabling technology to allow the Linux kernel based operating system to expose the self-aware adaptive behavior long discussed in this work.

Aim of this Chapter is to provide the reader with all the insight encountered to fully develop and implement the two methodologies previously proposed. As reported in the previous Chapter, the former methodology is complete and allows the implementation of self-aware adaptive applications and hence self-aware adaptive computing systems. The latter methodology is an enabling technology and hence a first step toward the integration of self-aware adaptive capabilities within a mainstream and widespread operating system kernel: the Linux kernel.

The remainder of this Chapter is organized as follows. Section 4.1 gives details on providing reconfiguration hardware devices support within in a streamlined operating system based on the Linux kernel. Section 4.2 describes the development process and the implementation of a self-aware adaptive library for the chosen case study: a cryptographic application. Section 4.3 propose a further refined approach with respect to the one proposed in Section 4.2 a different case study has been taken into account: a cryptographic hash application. Finally, the implementation of stacked Loadable kernel modules (LKMs) is described in Section 4.4 highlighting the key points allowing the creation of a kernel mode hot-swap mechanism.

4.1 Provide reconfigurable hardware support within the operating system

The support for online reconfiguration for hardware components, which is usually known as dynamic reconfiguration in the reconfigurable computing context, is provided at different levels, raging from a low-level, direct interaction with the reconfigurable hardware device to an high-level interface toward user mode applications. As a consequence, its actual implementation is structured into multiple components, which separate hardware-independent aspects from hardware-dependent aspects.

A schematic of the architecture of a computing system supporting dynamic reconfiguration is shown in Figure 4.1. Dynamic reconfiguration support is provided by means of three sub-components: the *Reconfiguration controller*, which is an hardware-dependent component, the *Reconfiguration device driver*, that is both operating system-dependent, since it has to conform to the operating system's interface (*i.e.,* to the Linux kernel interface) for device drivers, and hardware-dependent because of the "link" with certain kind of reconfigurable hardware architectures and reconfiguration controller, and finally the *Linux kernel interface* (*i.e.,* system calls' layer), which obviously is hardware-independent.

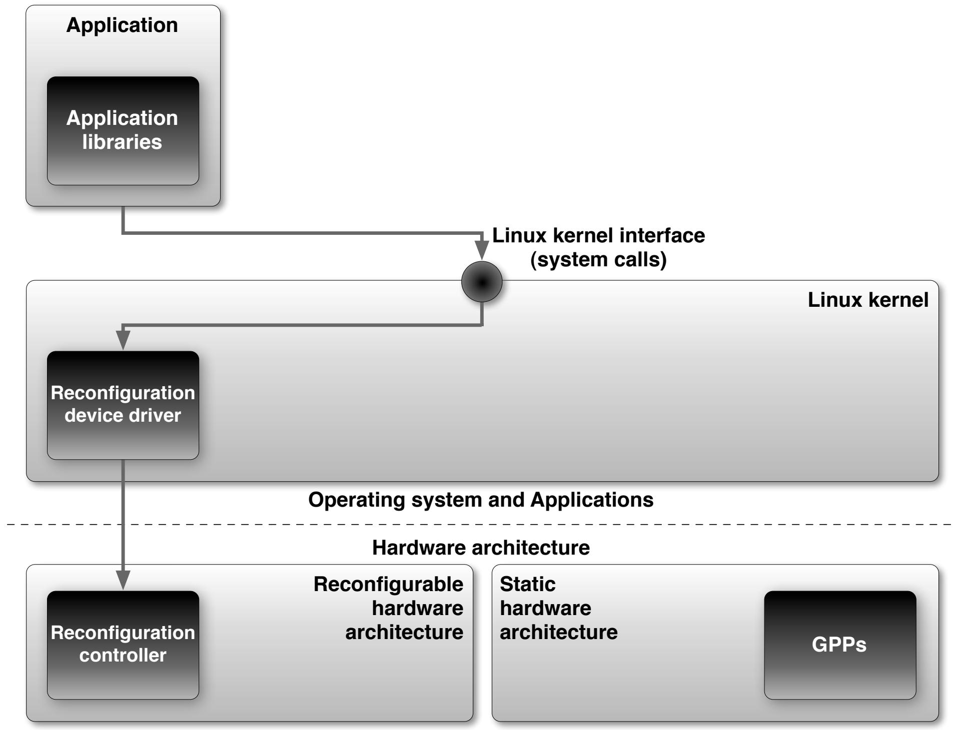


Figure 4.1: Computing system's components for dynamic reconfiguration support

The hardware architecture from Figure 4.1 can be implemented on a single Field Programmable Gate Array (FPGA) as a System-on-Chip (SoC), hence the configured Intellectual Property Cores (IP-Cores) must share a common area. Moreover, the operating system is executed on a General Purpose Processor (GPP) which is built in the FPGA area too, and it is not coupled with any external machine[1]. As a consequence, the FPGA must provide an interface which allows partial, internal, dynamic reconfiguration, in order to reconfigure only a portion of the reconfigurable hardware device without the support of an external machine.

In order to support the dynamic reconfiguration of IP-Cores, the target hardware architecture is logically divided into two distinct sections: the static section and the reconfigurable section (*i.e.,* reconfigurable area). The static section is never altered by dynamic reconfiguration processes, and it contains both the logic components which are always required (*i.e.,* the GPP, the internal reconfiguration controller, and the modules which are connected to the external pins such as the memory controller, the serial communication controller, the network communication, and so on). The reconfigurable area can be exploited to configure additional IP-Cores, and communication between the two sides is resolved according to the specific design flow used in the design phase.

Dynamic reconfiguration support is implemented by means of a LKM, actually a device driver, and an interface toward this LKM provided by the operating system based on the Linux kernel. The reconfiguration controller device driver is in charge of configuring the requested IP-Core on the reconfigurable area, while the operating system provides access to the configured IP-Core by means of an ad-hoc device driver implemented by a LKM. The interface toward the LKM handling the dynamic reconfiguration

[1] An external machine is required to provide a command line interface toward the final user, but it does not play an active role in the reconfiguration support.

is used drive the dynamic reconfiguration from user mode.

The reconfiguration device driver is a LKM that communicates with the physical component, the reconfiguration controller, that performs dynamic reconfiguration on the FPGA. Such a component is named in different ways according to the different vendors; for instance, the reconfiguration controller on Xilinx FPGAs is called Internal Configuration Access Port (ICAP), and it can be accessed within the reconfigurable hardware device itself to perform internal reconfiguration. The reconfiguration controller is employed to write an IP-Core, in form of its configuration code (*i.e.*, bitstream) on the reconfigurable hardware device, or to read the actual configuration.

Every configurable IP-Core must come with a LKM working as its device driver. The device driver is loaded inside the operating system kernel and is passed the unique physical address assigned to the mapped IP-Core to allow the usage of the IP-Core as it is a "normal" peripheral.

Being a device driver for the reconfiguration controller, the reconfiguration device driver exposes by means of the system calls' layer a set of functions (*e.g.*, `open()`, `read()`, `write()`, `close()`, and so on) allowing user mode processes to drive the dynamic reconfiguration process.

The reconfiguration controller we are using is the ICAP [18] provided by Xilinx for its FPGAs and the reconfiguration device driver to access dynamic reconfiguration from user mode processes is available in the mainline of the Linux kernel from version 2.6.25[2].

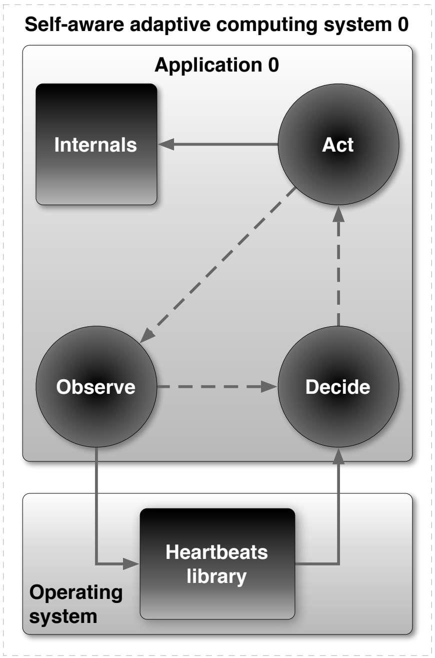
[2]The reconfiguration device driver is available under the directory `drivers/char/xilinx_hwicap/` of the Linux kernel sources.

4.2 Case study: a cryptographic self-aware adaptive library

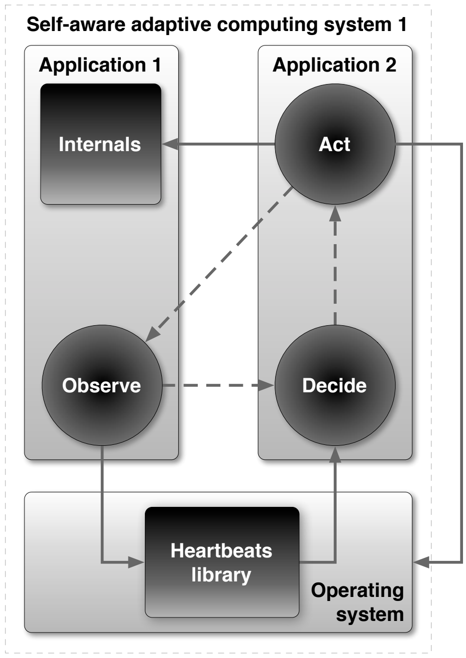
The support for dynamic reconfiguration previously described provides a complete method to configure an IP-Core on the FPGA and to interact with it. Each user application can use the capabilities of the reconfiguration controller to place the requested hardware implementation on the reconfigurable hardware device. Having bolstered the capabilities and the flexibility of our computing system thanks to reconfigurable hardware and dynamic reconfiguration, we can now use the hot-swap mechanism describe in the previous Chapter providing implementation libraries shaped as Dynamic-Link Librarys (DLLs) for both pure-software and hardware implementations.

We chose to implement a cryptographic self-aware adaptive library to perform both encryption and decryption using Data Encryption Standard (DES) [41]. As described in Section 3.2 there are two different scenario in building self-aware adaptive libraries using Heartbeats [42], they are shown in Figure 4.2a, in which a self-aware adaptive application - *Application 0* - is self-contained since it performs every step of the Observe, Decide, Act (ODA) loop by itself and in Figure 4.2b, in which a self-aware adaptive application - *Application 1* - is controlled by means of an external entity - *Application 2* in the example - that can either be another application or the operating system itself. For this case study our choice was to adopt the former scenario represented in Figure 4.2a.

The first operation to do consisted in choosing the set of implementations performing both encryption and decryption using DES, eventually modifying them to obtain a set of implementations in form of DLLs. We provided both a software implementation and an hardware implementation. The software implementation of DES exposes the set of functions re-



(a)



(b)

Figure 4.2: Self-aware adaptive computing systems using Heartbeats: (a) Application 0 is a self-contained self-aware adaptive application acts on its own internals to adjust the way it is working; (b) Application 1 is a self-aware adaptive application whose internals are controlled by means of an external entity which can be another application as Application 2 does in the example or the operating system itself which is in control not only of Application 1's internals but also of computing system's internals

Table 4.1: DES software library API

`uint64_t sw_des_encrypt(...)`		
`IN`	`uint64_t pt`	plain text
`IN`	`uint64_t ke`	encryption key
`OUT`	`uint64_t`	returns cipher text
`uint64_t sw_des_decrypt(...)`		
`IN`	`uint64_t ct`	cipher text
`IN`	`uint64_t kd`	decryption key
`OUT`	`uint64_t`	returns plain text

ported in Table 4.1. The hardware implementation, exploiting the dynamic reconfiguration capabilities described above, consists of an IP-Core which is accessed by means of the system calls' layer of the operating system based on the Linux kernel. These system calls are performed thanks to a device driver designed to fit the Linux kernel interface exposing the following functions: `open()`, `ioctl()`, `write()`, `read()`, and `close()`. The `write()` function is used to provide the IP-Core with both the key and either the plain text or the cipher text. The device driver is instructed on the data it should expect by the `ioctl()` function, which is also used to set the operation (*i.e.*, encrypt or decrypt) the IP-Core must perform. The `read()` function is used to get the results from the IP-Core. Both the `write()` function and the `read()` function are parametric, therefore, they can accept a variable number of blocks to operate on. A DLL wrapping these basic interface was developed to expose a "simpler" interface and to allow the self-aware adaptive library to be linked against the library providing the hardware implementation. The hardware implementation of DES exposes the set of functions reported in Table 4.2.

Table 4.2: DES hardware library API

`int hw_des_key(...)`		
IN	`uint64_t k`	encryption/decryption key
OUT	`int`	returns 0 on success, returns -1 and sets `errno` on failure
`int hw_des_encrypt(...)`		
IN IN	`uint64_t *pt` `size_t nb`	pointer to plain text number of block of both plain and cipher text
OUT OUT	`uint64_t *ct` `int`	pointer to cipher text returns 0 on success, returns -1 and sets `errno` on failure
`int hw_des_decrypt(...)`		
IN IN	`uint64_t *ct` `size_t nb`	pointer to cipher text number of block of both plain and cipher text
OUT OUT	`uint64_t *pt` `int`	pointer to plain text returns 0 on success, returns -1 and sets `errno` on failure

The implementation of the self-aware adaptive library requires the definition of a single and consistent interface to expose the realized functionality. The cryptographic self-aware adaptive library exposes the set of functions reported in Table 4.3.

4.2.1 Observe: Heartbeats

Having defined the interface of the self-aware adaptive library, the next step consists in instrumenting such library with Heartbeats, the observation sub-system we chose, to allow performance monitoring. Instrumenting an application making use of a monitoring framework may seem a straightforward task, however, there are some key aspects that must be taken into account. Heartbeats comes with two benchmarking applications to measure both *latency* and *throughput* of the monitoring framework on a certain computing system. These measures give important hints on Heartbeats usage in writing self-aware adaptive applications. After running these benchmarking applications on a variety of computing systems (*i.e.,* from embedded to notebook and desktop), we found a satisfying compromise in using the Heartbeat to avoid en excessive overhead of the monitoring framework on all those computing systems.

The pseudo-code reported in Listing 4.1 shows the usage of Heartbeats within a function exposed by the self-aware adaptive library. It is important to note the use of the `B` constants, at line 9, 18, and 19, which is the result of the *throughput* analysis. `B` indicates how many blocks per *heartbeat* the library must compute and it is also important because it indicates the minimum amount of blocks the library computes before deciding if the current implementation adopted to provide the exposed functionality is performing as expected or not.

Table 4.3: DES self-aware adaptive library API

int des_encrypt(...)		
IN	uint64_t *pt	pointer to plain text
IN	size_t nb	number of block of both plain and cipher text
IN	uint64_t ke	encryption key
IN	double m_hr	minimum heart rate
IN	double M_hr	maximum heart rate
OUT	uint64_t *ct	pointer to cipher text
OUT	int	returns 0 on success, returns -1 and sets errno on failure

int des_decrypt(...)		
IN	uint64_t *ct	pointer to cipher text
IN	size_t nb	number of block of both plain and cipher text
IN	uint64_t ke	decryption key
IN	double m_hr	minimum heart rate
IN	double M_hr	maximum heart rate
OUT	uint64_t *pt	pointer to plain text
OUT	int	returns 0 on success, returns -1 and sets errno on failure

```
int des_encrypt(...)
{
        int cb = 0;             // computed blocks
        int ai = SW;            // active implementation
        ...
        while (cb < nb) {
                ...
                if (ai == SW) {
                        for (int i = 0; i < (B < nb ? B : nb); ++i) {
                                *(ct + cb) = sw_des_encrypt(pt + cb,
                                        ke);
                                ++cb;
                        }
                } else if (ai == HW) {
                        dr();   // dynamic reconfiguration if needed
                        hw_des_key(ke);
                        hw_des_encrypt(pt + cb, ct + cb,
                                B < nb - cb ? B : nb - cb);
                        cb += B < nb - cb ? B : nb - cb;
                }
                heartbeat(...);
        }
        ...
        return 0;
}
```

Listing 4.1: Observe using Heartbeats

4.2.2 Decide: heuristic model

As reported in Section 3.3, the use of empirical models alongside with Heartbeats may result in a winning approach. Since developing an accurate learning mechanism and a sophisticated decision mechanism is beyond the scope of this thesis work, we develop a simple yet effective decision mechanism based on an heuristic. The adoption of such heuristic has the advantage of limiting the overhead of the decision mechanism on the execution of self-aware adaptive applications just like Heartbeats do for the observation mechanism. A sketch of the implementation is shown in the following pseudo-code, Listing 4.2.

```
1  uint64_t m_s_tw;                 // minimum switch time window
2  ...
3  int des_encrypt(...)
4  {
5          int cb = 0;              // computed blocks
6          int ai = SW;             // active implementation
7          uint64_t c_ts;           // current time stamp
8          uint64_t l_s_ts;         // last switch time stamp
9          double w_hr;             // window heart rate
10         ...
11         while (cb < nb) {
12                 c_ts = get_current_time_stamp();
13                 if (c_ts - l_s_ts > m_s_tw) {
14                         w_hr = hb_get_windowed_rate(...);
15                         ...
16                         if (w_hr < m_hr || w_hr > M_hr ...) {
17                                 if (ai == SW) {
18                                         ai = HW;
19                                 } else if (ai == HW) {
20                                         ai = SW;
21                                 }
22                                 l_s_ts = c_ts;
23                         }
24                 }
25                 ...
26         }
27         ...
28         return 0;
29 }
```

Listing 4.2: Decision using Heartbeats

In the implementation sketch shown in Listing 4.2, it is important to note the usage of Heartbeats to get the current *window heart rate* (*i.e.*, `w_hr`), at line 14, representing the performance of the application till the last heartbeat sent. This value is compared, at line 16, against two values: the *minimum heart rate* (*i.e.*, `m_hr`) and the *maximum heart rate* (*i.e.*, `M_hr`); these two values are the two parameter defining the requested Quality-of-Service (QoS). In addition to this, the piece of pseudo-code reported makes use of the actual time stamp to guarantee no excessive switches occur between the two available implementation, realizing one of the goal of the simple heuristic proposed in Section 3.3.

4.2.3 Act: the implementation switch service

The two implementations adopted to build the cryptographic self-aware adaptive library have been designed to allow the simplest implementation of the hot-swap mechanism. In fact, even tough they don't expose the exact same interfaces and they don't use the exact same data structures, their data structure are pretty similar in the basic data types they use. Therefore, the translation from the "canonical" data structure chosen for the implementation of the self-aware adaptive library to both the data structures of the two implementations is straightforward. The hot-swap mechanism is shown in Figure 4.3.

In Figure 4.3a the "canonical" data structure is translated into the data structure suiting the software implementation and the function call starts. Upon return of the function call, the data structure of the software implementation is again translated, this time to "canonical" data structure. In Figure 4.3b the same process is done making use of the hardware implementation and hence of its data structure. This *on-the-fly* translation allows the self-aware adaptive library to switch between the two available implementations multiple times per function call issued by the self-aware adaptive application.

The hot-swap mechanism implemented is made up of fewer steps than the *state-of-the-art* approach presented in Section 2.3. The number of steps reduction is due to a slightly different approach. While the general approach takes in consideration the adoption of self-contained switchable

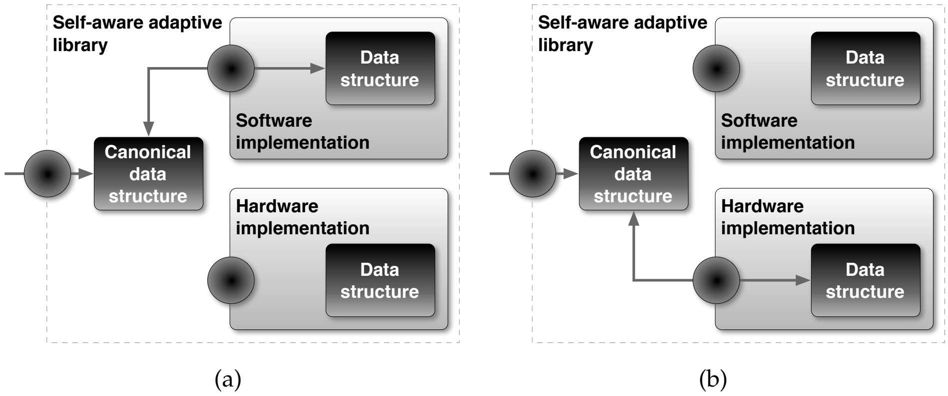


Figure 4.3: Hot-swap process. This Figure shows the phases of the hot-swap process between two different implementation with different data structures. When the self-aware adaptive library is using the software implementation the "canonical" data structure exposed by the self-aware adaptive library is translated into the data structure suiting the software implementation. Otherwise, when the self-aware adaptive library is using the hardware implementation the "canonical" data structure is translated into the data structure suiting the hardware implementation

unit, we decide to build our self-aware adaptive library on pre-existing components, in this case DLLs which are not self-contained switchable unit as one might intend. The results is a self-aware adaptive libraries acting as wrapper between DLLs, each of each providing a different implementation of the same functionality, and the self-aware adaptive applications making use of such library. Since DLLs are already linked against the self-aware adaptive library, the hot-swap mechanism needs to concentrate on the state translation problem since it has to work solely on data structures. The translation problem is solved in the chosen case study using *on-the-fly* translation because of the data structures involved are similar, otherwise, the factory design pattern [43] may result a winning approach to solve such problem in more complex situations.

4.3 Case study: a cryptographic hash self-aware adaptive library

In addition to the implementation of the DES self-aware adaptive library described in the previous Section we developed a further refined approach that makes use of a different platform and more advanced techniques to build the self-aware adaptive library.

The same hardware architecture from Figure 4.1 can be mapped on an heterogeneous architecture where the static hardware architecture is a "standard" computing system while the reconfigurable hardware architecture is the whole FPGA which is now used as an accelerator interfaced to the computing system with as system bus such as the Peripheral Component Interconnect (PCI) Express. In this case the FPGA is not required to support partial, internal reconfiguration since en external reconfiguration controller can be used and the whole area can be re-programmed.

Having both a GPP and an FPGA we exploited both of them and provided both a software and an hardware implementation and we chose to implement a self-aware adaptive library exposing the Secure Hash Algorithm 1 (SHA1) cryptographic hash function [44, 45]. For this case study too our choice was to adopt Heartbeats following the scenario reported in Figure 4.2a.

This time we defined a common interface for both the software and the hardware implementation of SHA1; the common interface is reported in Table 4.4. Both the software and the hardware implementation libraries conform to the same common interface and export the same symbols. As required by the proposed methodology each implementation library is packed as a DLL. Once again, the hardware implementation library works a wrapper for an ad-hoc IP-Core which is accessed by means of the system calls layer of the operating system based on the Linux kernel. These system calls

Table 4.4: SHA1 common API

`void sha1_init(...)`		
`IN`	`struct sha1_context *ctx`	SHA1 context
`int sha1_compute(...)`		
`IN`	`struct sha1_context *ctx`	SHA1 context
`IN`	`const uint8_t m[]`	input message
`IN`	`size_t len`	input message size
`OUT`	`int`	returns either `EXIT_SUCCESS` or `EXIT_FAILURE`
`int sha1_compute_final(...)`		
`IN`	`struct sha1_context *ctx`	SHA1 context
`IN`	`const uint8_t m[]`	input message
`IN`	`size_t len`	input message size
`OUT`	`int`	returns either `EXIT_SUCCESS` or `EXIT_FAILURE`
`int sha1_exit(...)`		
`OUT`	`const uint8_t h[]`	hash
`IN`	`struct sha1_context *ctx`	SHA1 context
`OUT`	`int`	returns either `EXIT_SUCCESS` or `EXIT_FAILURE`

Table 4.5: SHA1 self-aware adaptive library API

int sha1_saa(...)		
OUT	const uint8_t h[]	hash
IN	const uint8_t m[]	input message
IN	size_t len	input message size
IN	double m_hr	minimum heart rate
IN	double M_hr	maximum heart rate
OUT	int	returns either EXIT_SUCCESS or EXIT_FAILURE

are performed thanks to a device driver designed to fit the Linux kernel interface exposing the following functions: `open()`, `write()`, `read()`, and `close()`. The `write()` function is used to provide the IP-Core with both the context and the message while the `read()` function is used to get back the updated context.

The self-aware adaptive library requires the definition of a single and consistent interface to expose the realized functionality. The cryptographic hash self-aware adaptive library exposes the function reported in Table 4.5.

One again, after defining the interface the self-aware adaptive library exposes, the next step consists in instrumenting such library with Heartbeats, the observation sub-system we chose, to allow performance monitoring. As reported for the previous case study the instrumentation depends on the results coming from the *throughput* benchmark packed with Heartbeats. The pseudo-code reported in Listing 4.1 shows an example of how Heartbeats should be used within the function exposed by the self-aware adaptive library. This revised implementation of the user space hot-swap mechanism does not affect how the observation and the decision phase are executed.

What changes with this modified approach is how the hot-swap mechanism is actually implemented and hence how the act phase is executed. In the previous case study the hot-swap mechanism consisted in taking a code path instead of another and eventually translates data structures used by the active implementation library to fit the new implementation library. However, an improved version of such a mechanism, that is more biased towards online reconfiguration, can be implemented and is proposed below.

4.3.1 Act: the improved hot-swap mechanism

Having defined a consistent interface between the software and the hardware implementation library we can build a better and more general revision of the hot-swap mechanism using function pointers and dynamic code load. At the very base of this new approach is the adoption of a data structure containing a set of function pointers which are meant to refer to pieces of code that are dynamically loaded from implementation libraries on-demand. This allows the implementation of the self-aware adaptive library to be simpler since no branches are needed to actually differentiate the code path referring to one implementation library with respect to the code path of referring to another implementation library. In addition to this, adding a new implementation library is way simpler since we only need to "inform" the self-aware adaptive library that another implementation is available.

The hot-swap mechanism implemented following this improved approach is based on the *state-of-the-art* framework discussed in Section 2.3 since it requires to stop the current execution, atomically switch between the active implementation library to the new implementation library and restart the execution. Listing 4.3 shows the pseudo-code of the improved hot-swap mechanism with the basic data structure featuring all the necessary function pointers referring to the active implementation library, the hot-swap function, and the function that exports the functionality from the self-aware adaptive library.

```c
struct sha1_impl {
	...
	void (*sha1_init)(struct sha1_context *);
	int (*sha1_compute)(struct sha1_context *, const uint8_t [], size_t);
	int (*sha1_compute_final)(struct sha1_context *, const uint8_t [], size_t);
	int (*sha1_exit)(uint8_t [], struct sha1_context *);
	void *so_handler;
	...
};

int sha1_saa(...)
{
	...
	sha1_impl_init();
	...
	struct sha1_context ctx;
	(*impl_current.sha1_init)(&ctx);
	...
	size_t i = 0;
	for (; i < b; i += B) {
		(*impl_current.sha1_compute)(&ctx, m + i, B);
		heartbeat(...);
		double w_hr = hb_get_windowed_rate(...);
		if (w_hr < m_hr || hr_windowed > M_hr) {
			sha1_impl_swap(...);
		}
	}
	(*impl_current.sha1_compute_final)(&ctx, m + i, B);
	i += B;
	heartbeat(...);
	double hr_windowed = hb_get_windowed_rate(...);
	if (w_hr < m_hr || w_hr > M_hr) {
		sha1_impl_swap(...);
```

```
        }
        (*impl_current.sha1_exit)(h, &ctx);
        ...
}

int sha1_impl_swap(...)
{
        ...
        struct sha1_impl impl_new;
        impl_new.so_handler = so_open(...);
        *(void **) (&impl_new.sha1_init) =
                so_load_sym(impl_new.so_handler, SHA1_INIT);
        *(void **) (&impl_new.sha1_compute) =
                so_load_sym(impl_new.so_handler, SHA1_COMPUTE);
        *(void **) (&impl_new.sha1_compute_final) =
                so_load_sym(impl_new.so_handler, SHA1_COMPUTE_FINAL);
        *(void **) (&impl_new.sha1_exit) =
                so_load_sym(impl_new.so_handler, SHA1_EXIT);
        so_close(impl_current.so_handler);
        memset(impl_current, impl_new, sizeof(struct sha1_impl);
        ...
}
```

Listing 4.3: Improved hot-swap mechanism

The function calls `so_open`, `so_load_sym`, and `so_close` wrap the dynamic linking functions[3].

4.4 Case study: a stacked LKM featuring two different implementations

Most of the UNIX kernels are monolithic and the first versions of the Linux kernel were considered monolithic too, until the introduction of LKMs in version 1.2 that gave birth to a monolithic kernel that is both scalable and dynamic [46]. In monolithic kernel each layer is integrated into a single "big" program that runs in the so called kernel mode in behalf of the

[3]Dynamic linking functions are defined in the header file `dlfcn.h`.

current process. On the other side, micro-kernels demand a little set of function running in kernel mode, generally a basic process scheduler, a memory manager, and basic Inter-Process Communication (IPC) mechanisms while the other fundamental parts of the operating system (*e.g.,* mostly device drivers) are implemented by means of user mode processes.

As a general note, academic research is biased toward microkernels while mainstream kernels usually follow the monolithic approach. Microkernels may have some theoretical advantage in terms of modularization which is enforced by the model itself. These advantages may be: portability, since the amount of kernel mode code to be modified consists of few lines of code; resources usage, because system processes that are not in use can be swapped out or shut down. However, microkernels generally suffer of performance issue due to the overhead needed to execute most of the operations in user mode.

To achieve many of the advantages of microkernels without introducing performance penalties, the Linux kernel can be dynamically altered at run time through the use of LKMs[4]. A LKM is an object file whose code can be linked to or unlinked from the kernel at run time. Being part of the Linux kernel, once they are loaded, LKMs are executed in kernel mode on behalf of the current process just like the statically linked code.

As reported in [35] LKMs cannot modify an already defined data structure; even if a LKM uses its own modified version of the data structure, all the statically linked code continues to see the old version of the data structure since changing all the references is a non-feasible process. However, it must be noted that modifying existing data structure is possible whenever a method to change the running code (*i.e.,* modifying the statically linked

[4]The Linux kernel is not the only "monolithic" kernel providing LKMs. Loadable kernel modules can be found also in BSD variants, Oracle Solaris Operating System, OpenSolaris, Microsoft Windows operating system, and in other kernel that can be defined "hybrid" like the one of Apple Mac OS X.

code residing within the main memory) is adopted as reported in [47].

As reported in Section 3.4.2, to implement an hot-swap mechanism for the LKMs within the Linux kernel, we though to modify the structure of a LKM following the winning approach of Clustered Objects (COs). Therefore, we bi-partition every LKM that wants to exploit the hot-swap mechanism in a front end LKM and a back end LKM. The front end LKM is in charge to expose the functionality through the Linux kernel interface, while the back end LKM is meant to implement the functionality complying with a pre-defined interface between the front end LKM and the back end LKM. The back end LKM exports such an interface by means of a set of *kernel symbols* registered through two macros: `EXPORT_SYMBOL` and `EXPORT_SYMBOL_GPL`; the former macro is used to export symbols available to every entity within the Linux kernel, while the latter is used to export symbols available to General Public License (GPL) entities only. Every front end LKM is free to define its interface to back end LKMs best suiting its needs, there is no constraint the hot-swap mechanism impose.

For the first sketched implementation of the stacked LKMs we require every back end LKM to use the same data structure which is handled by the front end LKM allowing a simpler implementation of the hot-swap mechanism. This may seem a tough restriction, however, using opaque data structures it is possible to obtain usable data structure even respecting this constraint. For future implementations of this hot-swap mechanism the front end LKM the *factory design pattern* [43] may result handy again, allowing the use of specialized data structures side by side of a straightforward translation functionality. However, the adoption of such a design does not remove the constraint on the positioning of the active data structure that must remain within the front end LKM boundaries.

The chosen case study regards a software driver to manage a simple character device. This character device is used as an interface to allocate memory, which is readable and writeable by user mode processes, within the kernel address space boundary. Key functions of this software driver are implemented in two ways by means of two back end LKMs interfacing with the front end LKM. There two back end LKMs implements respectively: a non-blocking policy and a blocking policy.

Listing 4.4 provides a draft implementation of the "generic" data structure, designed for both the non-blocking and the blocking implementation, and those shared functions initializing and de-initializing the LKM. The `cd_dev` data structure, defined at line 1, is opaque in the sense it is valid for both the implementation of the software driver. Moreover, it is important to note that both the implementations must update even those fields of the data structure they do not use to allow a fast and transparent switch between the available implementations.

```
1  struct cd_dev {
2          void *data;
3          unsigned int data_size;
4          unsigned int size;
5          unsigned long r_pos;
6          unsigned long w_pos;
7          struct semaphore sem;
8          wait_queue_head_t r_queue;
9          struct cdev cdev;
10 };
11
12 static struct cd_dev *dev;
13 ...
14 static struct file_operations fops = {
15         .owner = THIS_MODULE,
16         .read = cd_read_front,
17         .write = cd_write_front,
18         .llseek = cd_llseek,
19         .open = cd_open,
20         .release = cd_release
21 };
22 ...
23 static int __init cd_init(void)
```

```
{
        dev_t dev_id = 0;
        ...
        alloc_chrdev_region(&dev_id, 0, 1, "cd");
        M_num = MAJOR(dev_id);

        dev_list = kzalloc(sizeof(struct cd_dev), GFP_KERNEL);
        init_waitqueue_head(&dev->r_queue);
        init_MUTEX(&dev->sem);
        cdev_init(&dev->cdev, &fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &fops;
        cdev_add(&dev->cdev, dev_id, 1);
        ...
        return 0;
}
module_init(cd_init);

static void __exit cd_exit(void)
{
        dev_t dev_id = MKDEV(M_num, 0);

        if (dev) {
                kfree(dev->data);
                cdev_del(&devs->cdev);
                kfree(dev);
        }

        unregister_chrdev_region(dev_id, 1);
}
module_exit(cd_exit);
```

Listing 4.4: Front end LKM: “generic” data structure and shared functions

Listing 4.5 shows the pseudo-code of those functions whose implementation is shared, and therefore provided by the front end LKM, between the two different back end LKMs.

```
static int cd_open(struct inode *inode, struct file *filp)
{
        struct cd_dev *dev;
```

```

        dev = container_of(inode->i_cdev, struct cd_dev, cdev);
        filp->private_data = dev;

        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        if ((filp->f_flags & O_ACCMODE) == O_WRONLY) {
                kfree(dev->data);
                dev->data_size = 0;
                dev->size = 0;
                dev->r_p = 0;
                dev->w_p = 0;
        }

        up(&dev->sem);
        return 0;
}

static int cd_release(struct inode *inode, struct file *filp)
{
        return 0;
}

static loff_t cd_llseek(struct file *filp, loff_t f_pos, int whence)
{
        struct cd_dev *dev = filp->private_data;
        loff_t new_f_pos;

        if (whence == SEEK_SET) {
                new_f_pos = f_pos;
        } else if (whence == SEEK_CUR) {
                new_f_pos = filp->f_pos + f_pos;
        } else if (whence == SEEK_END) {
                new_f_pos = dev->data_size + f_pos;
        } else {
                return -EINVAL;
        }

        if (new_f_pos < 0)
                return -EINVAL;
        if (new_f_pos >= device->data_size)
```

```
                return -EINVAL;
        filp->f_pos = new_f_pos;
        dev->r_pos = new_f_pos;
        dev->w_pos = new_f_pos;
        return new_f_pos;
}
```

Listing 4.5: Front end LKM: shared functions

Listing 4.6 displays the pseudo-code of those functions whose implementation is both shared between the two different "personalities" of the software driver and dependent from the set of kernel symbols exported within the two different back end LKMs. These functions whose behavior depends on the linked implementation (*i.e.,* `cd_read_front` and `cd_write_front`) are implemented using the kernel symbols exported by the back end LKM (*i.e.,* `cd_read_back` and `cd_write_back`, used respectively at line 10 and 39).

```
1  static ssize_t cd_read_front(struct file *filp, char __user *buf,
2                  size_t len, loff_t *f_pos)
3  {
4          struct cd_dev *dev = filp->private_data;
5          ssize_t ret_val = 0;
6
7          if (down_interruptible(&dev->sem))
8                  return -ERESTARTSYS;
9
10         ret_val = cd_read_back(filp, buf, len, f_pos, dev);
11
12         up(&dev->sem);
13         return ret_val;
14 }
15
16 static ssize_t cd_write_front(struct file *filp,
17         const char __user *buf, size_t len, loff_t *f_pos)
18 {
19         struct cd_dev *dev = filp->private_data;
20         void *data;
21         unsigned int size;
```

```
        ssize_t ret_val = 0;

        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;

        if (!dev->data) {
                dev->size = ((*f_pos + len) / PAGE_SIZE + 1) *
                        PAGE_SIZE;
                dev->data = kzalloc(dev->size, GFP_KERNEL);
        } else if (*f_pos + len > dev->size) {
                size = ((*f_pos + len) / PAGE_SIZE + 1) * PAGE_SIZE;
                data = kzalloc(size, GFP_KERNEL);
                memcpy(data, dev->data, dev->data_size);
                kfree(dev->data);
                dev->data = data;
                dev->size = size;
        }

        ret_val = cd_write_back(filp, buf, len, f_pos, dev)

        return ret_val;
}
```

Listing 4.6: Front end LKM: policy dependent shared functions

Listing 4.7 shows the pseudo-code of the two functions exported as kernel symbols within the back end LKM implementing the non-blocking policy.

```
static ssize_t cd_read_back(struct file *filp, char __user *buf,
        size_t len, loff_t *f_pos, struct cd_dev *dev)
{
        if (!dev->data || *f_pos >= dev->data_size)
                return 0;

        len = min(len, dev->data_size - (size_t) *f_pos);
        copy_to_user(buf, dev->data + *f_pos, len);
        *f_pos += len;
        dev->r_pos = *f_pos;
        dev->w_pos = *f_pos;

```

```
        return len;
}
EXPORT_SYMBOL_GPL(cd_read_back);

static ssize_t cd_write_back(struct file *filp, const char
        __user *buf, size_t len, loff_t *f_pos, struct cd_dev *dev)
{
        copy_from_user(dev->data + *f_pos, buf, len);
        *f_pos += len;
        dev->data_size = *f_pos;
        dev->r_pos = *f_pos;
        dev->w_pos = *f_pos;

        up(&dev->sem);
        return len;
}
EXPORT_SYMBOL_GPL(cd_write_back);
```

Listing 4.7: Back end LKM: non-blocking policy

Listing 4.8 shows the pseudo-code of the two functions exported as kernel symbols within the back end LKM implementing the blocking policy.

```
static ssize_t cd_read_back(struct file *filp, char __user *buf,
        size_t len, loff_t *f_pos, struct cd_dev *dev)
{
        while (dev->r_pos >= dev->w_pos) {
                up(&device->sem);

                if (filp->f_flags & O_NONBLOCK)
                        return -EAGAIN;

                if (wait_event_interruptible(dev->r_queue,
                                (dev->r_pos < dev->w_pos)))
                        return -ERESTARTSYS;

                if (down_interruptible(&dev->sem))
                        return -ERESTARTSYS;
        }

        len = min(len, (size_t) (dev->w_pos - dev->r_pos));
        copy_to_user(buf, dev->data + dev->r_pos, len);
```

```
        dev->r_pos += len;

        return len;
}
EXPORT_SYMBOL_GPL(cd_read_back);

static ssize_t cd_write_back(struct file *filp, const char
        __user *buf, size_t len, loff_t *f_pos, struct cd_dev *dev)
{
        void *data;
        unsigned int size;

        copy_from_user(dev->data + dev->w_pos, buf, len);
        dev->w_pos += len;
        dev->data_size = dev->w_pos;
        *f_pos = dev->w_pos;

        up(&dev->sem);
        wake_up_interruptible(&dev->r_queue);
        return len;
}
EXPORT_SYMBOL_GPL(cd_write_back);
```

Listing 4.8: Back end LKM: blocking policy

The front end LKM is accessed as usual through the system calls' layer the Linux kernel provides, in addition, it is meant to be linked when the exported functionality is needed and unlinked when it is not needed anymore, hence, it is not modified by the hot-swap mechanism. Since the back and LKMs are not directly accessible by the user mode applications because of the "private" interface exported toward the front end LKM, they are registered with the Linux kernel only through the Kernel Symbol Table (KST), nonetheless, they are simpler to remove than "normal" LKMs. Since back end LKMs are accessible only through the front end LKM, the front end LKM acts as the *mediator object* the K42 operating system uses to accomplish the hot-swap of a CO with another one. The front end LKM can establish a quiescent state for the linked back end LKM in a straightforward way because of it is the only portion of the Linux kernel accessing the linked back end LKM. For example, the front end LKM's pseudo-code shown above can be simply modified to block function calls directed to the linked back end LKM allowing the swap-off of the linked back end LKM in favor of another back and LKM providing a different implementation.

LKM stacking is already in use in the mainstream Linux kernel sources: the *msdos* file system relies on symbols exported by the *File Allocation Table (FAT)* LKM, and each input Universal Serial Bus (USB) LKM stacks on the *usbcore* and *input* LKMs [48]. This has an important advantage on the proposed approach, the learning curve needed to develop stacked LKMs is really accessible for Linux kernel developers with experience in writing LKMs.

4.5 Summary

This Chapter first described the adoption of dynamic reconfiguration support within the operating system in order to use reconfigurable hardware devices. The operating system chose was one based on the Linux kernel which proved to be flexible enough to implement all the needed features.

Going further, this Chapter outlined the implementation of both a cryptographic and a cryptographic hash self-aware adaptive library to build self-aware adaptive applications following the adaptive code approach and the mixed adaptive code/online reconfiguration approach detailed in Chapter 3. These implementation are characterized by an high level of flexibility since they allow the re-use of pre-existing implementation libraries.

Finally, this Chapter delineated the implementation of a stacked LKM following the online reconfiguration approach reported in Chapter 3. The resulting LKM manages a memory region allocated within the Linux kernel address space and is provided with two different management policy implemented by means of two back end LKMs (*i.e.,* non-blocking and blocking). The use of LKMs stacking is the first enabling technology toward the implementation of an hot-swap mechanism within the Linux kernel.

The next Chapter will show the experimental results obtained using the cryptographic and the cryptographic hash self-aware adaptive libraries running respectively on an embedded computing system supporting dynamic reconfiguration and on an heterogeneous computing system featuring an FPGA connected through the PCI Express bus.

Chapter 5

Experimental results

In Chapter 4 we described the implementation of the proposed methodologies. In this Chapter we presents the testing platform and the experimental results to support the validity of the proposed approach.

The remainder of the Chapter is organized as follows: Section 5.1 gives a detailed description of the testing platform and Section 5.1.1 shows preliminary results collected in the chosen case study: a cryptographic application using the cryptographic self-aware adaptive library built following the implementation proposed in the previous Chapter.

5.1 Embedded computing system

Our implementation is meant to fit a wide range of computing systems, from mobile devices to cloud computing systems. We decided to adopt an embedded computing system supporting dynamic reconfiguration as testing platform to show how much our approach is flexible as reported in [40].

The testing platform is based on a *Xilinx University Program Virtex-II Pro (XUPV2P)* board featuring an XC2VP30 Field Programmable Gate Array (FPGA)-based board used to provide the basic architecture and Input/Output

(I/O) ports to run and interact with the operating system. The FPGA includes an *International Business Machines (IBM) PowerPC 405* Reduced Instruction Set Computer (RISC) processor [49] with a maximum frequency of 300 MHz coming with 16 kB of instruction cache and 16 kB of data cache [50]. The PowerPC 405 is used as the General Purpose Processor (GPP) of our embedded computing system and is allowed to access 256 MB of Synchronous Dynamic Random Access Memory (SDRAM) Double Data Rate (DDR)-266 used as main memory through the *Xilinx Multi-Port Memory Controller (MPMC)* SDRAM DDR controller [51]. On top of this architecture we loaded the *Linux kernel* version 2.6.33.1 (vanilla) [52] and a reduced root file system built upon *BusyBox* version 1.16.0 [53]. Both the Linux kernel and the root file system are compiled with a toolchain featuring *GNU Compiler Collection (GCC)* version 4.2.4 [54] and *uClibc* version 0.9.30.1 [55].

When the board is powered-on the bitstream implementing the architecture is taken from the first partition of the onboard compact flash memory (*i.e.,* the mass storage device) and used to program the FPGA while the Linux kernel and the root file system are taken from the second partition of the same mass storage device. The compact flash memory of 1 GB is accessible thanks to *Xilinx System ACE* storage controller [56].

The console of the Linux kernel is redirected to the *Xilinx Universal Asynchronous Receiver-Transmitter (UART) Lite* serial controller [57] and the embedded system is controlled by means of a *Telnet* session [58, 59] passing through the *Xilinx Ethernet Lite Media Access Controller (MAC)* network interface controller [60].

The architecture can be dynamically reconfigured at run time allowing the instantiation of an Intellectual Property Core (IP-Core) with a *Wishbone* [61] interface implementing Data Encryption Standard (DES) [41] encryption and decryption capabilities. This IP-Core is connected to the rest of the system by means of another IP-Core implementing a *Processor Local*

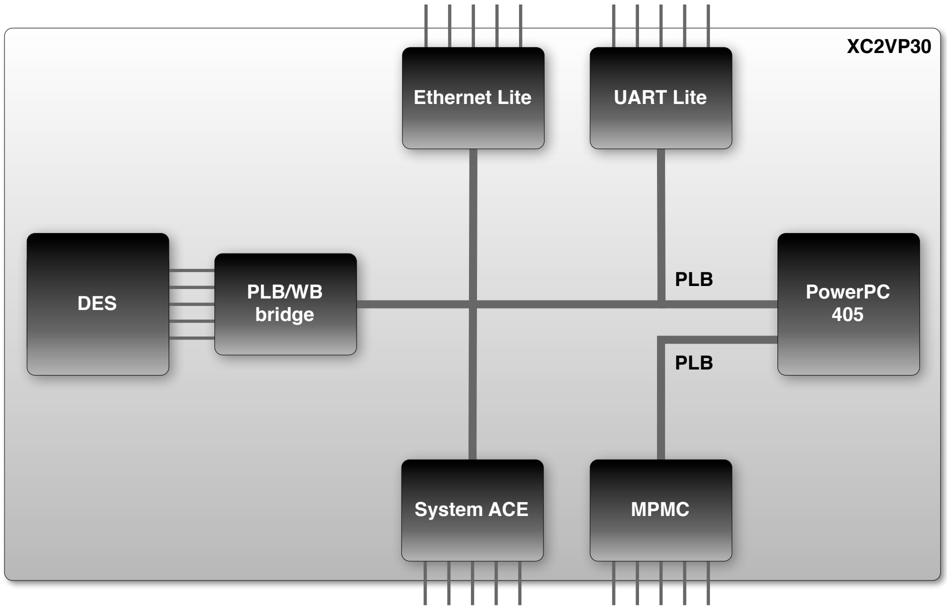


Figure 5.1: Block diagram of the implemented hardware architecture on a Xilinx XUPV2P featuring an XC2VP30 FPGA

Bus (PLB) [62] version 4.6 to Wishbone bridge. A schematics of the architecture is proposed in Figure 5.1.

5.1.1 Preliminary results and validation

It is indeed true that many problems are more efficiently solved using hardware implementations instead of software implementations. Yet this claim actually depends on the expected Quality-of-Service (QoS) meaning a software implementation might perform sufficiently well with respect to given constraints. Therefore we designed two DES implementations, one in software and one in hardware, provided by means of two Dynamic-Link Librarys (DLLs). Starting from these implementations, we develop a self-aware adaptive library in form of DLL following the approach specified in Section 4.2.

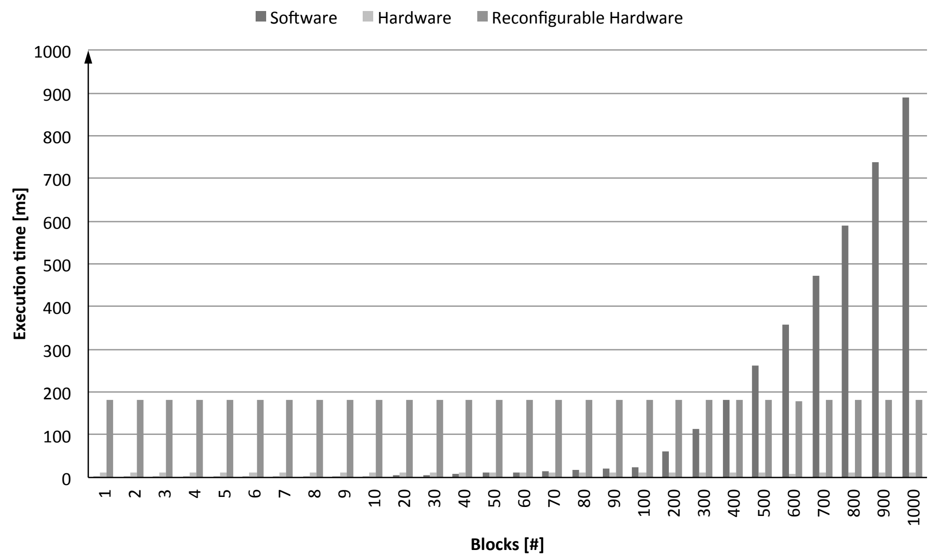


Figure 5.2: Execution times of the software implementation, the hardware implementation, and the reconfigurable hardware implementation to cipher 1 to 1000 blocks

Static analysis

The first result of our work is to statically analyze the ideal behavior of the two different implementations the self-aware adaptive library can adopt. We run both the implementations varying the input data size and averaging the results on tens of different trials; the hardware implementation was run in both configured state and non-configured state to take in consideration also the reconfiguration time. The results are shown in Figure 5.2. Let's see in more details what happens.

As shown in Figure 5.3, the software implementation (indicated in *dark grey*) outperforms the hardware implementation (indicated in *light gray*) for input data sizes inferior to 60 blocks (*i.e.,* 480 Byte). However, when the input data size gets bigger than or equal to 60 blocks the hardware implementation becomes faster than the software implementation.

Even though the hardware implementation is the clear winner when

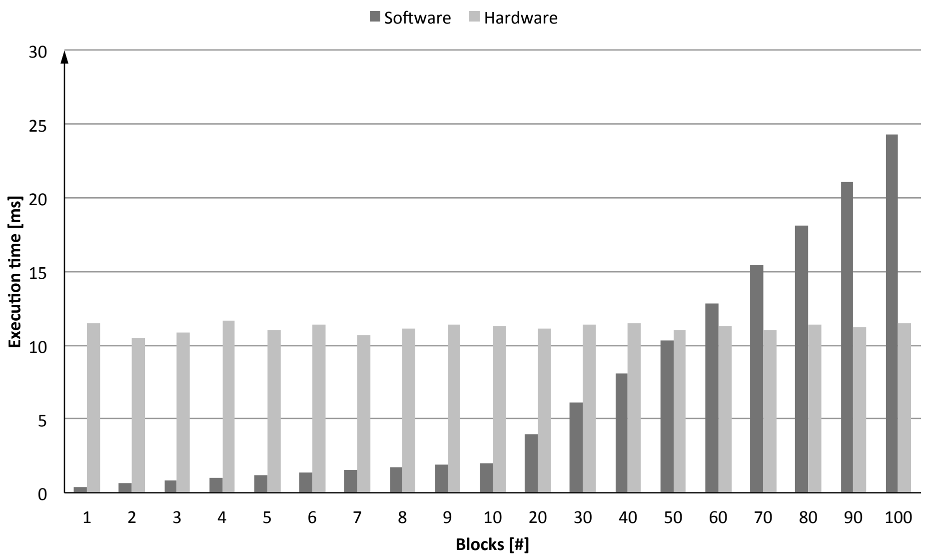


Figure 5.3: Execution times of the software implementation and the hardware implementation to cipher 1 to 100 blocks. The hardware implementation becomes faster when the input data size gets bigger than or equal to 60 blocks.

the input data size is greater than or equal to 60 blocks, Figure 5.4 shows that if the hardware implementation needs to be reconfigured on the reconfigurable area of the FPGA, a considerable amount of time, with respect to the evaluated execution time, needs to be spent. In fact, considering the reconfiguration time, the hardware implementation, now reconfigurable hardware implementation (indicated in *gray*), becomes competitive with respect to the software implementation only when the input data size gets bigger than 400 blocks (*i.e.,* 3200 Byte).

Figure 5.5 reveals an interesting trend over the ciphering of 1 to 1000 blocks. The execution time of the reconfigurable hardware implementation with respect to the execution time of the hardware implementation is dominated by the reconfiguration time. This is displayed by the overhead and the average overhead on the execution time. Moreover, the hardware implementation, and hence the reconfigurable hardware implementation,

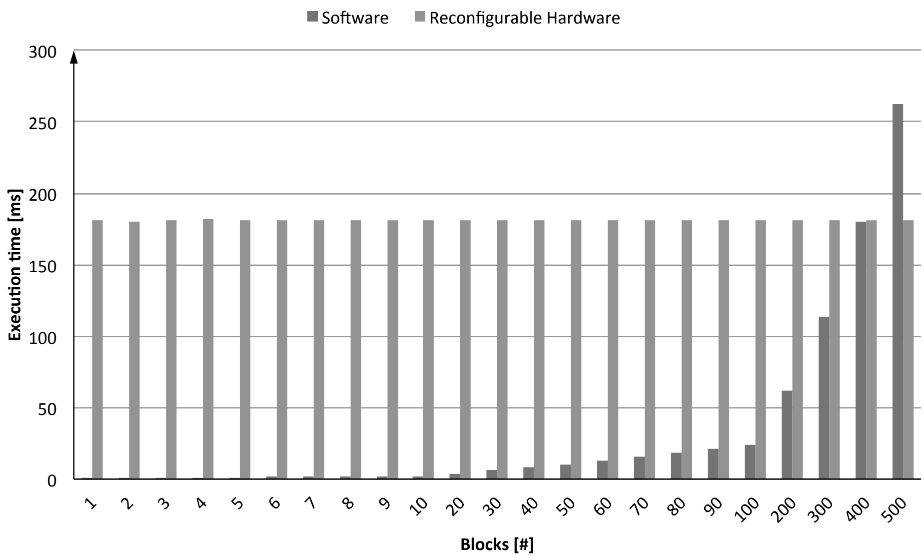


Figure 5.4: Execution times of the software implementation and the reconfigurable hardware implementation to cipher 1 to 400 blocks. The reconfigurable hardware implementation becomes competitive when the input data size gets bigger than or equal to 400 blocks.

puts on view an execution time which is practically constant, in fact, the execution time is dominated by both the context switch between the user mode and the kernel mode, and vice vera, and the data transfer between the buffer allocated by the user mode application and the buffer allocated by the kernel mode device driver; this happens every time the IP-Core is fed with blocks to cipher or decipher.

Dynamic analysis

The static analysis reported above might seem to prove that, depending on the input data size (*i.e.*, number of blocks), it is possible to choose the best implementation statically. Yet in a dynamic scenario such execution times might change completely due to the system load or constraints (*e.g.*, power consumption) and a statical approach might fail. Since plenty of factors cannot be predicted it is necessary to monitor the throughput of

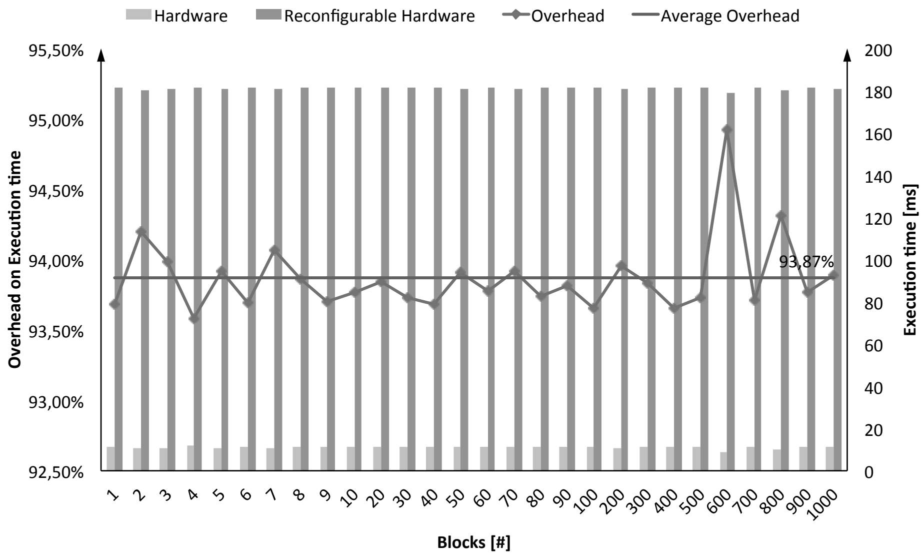


Figure 5.5: Execution times of the hardware implementation and the reconfigurable hardware implementation to cipher 1 to 1000 blocks. The reconfigurable hardware execution is clearly dominated by the reconfiguration time as the overhead and the average overhead on the execution time show

the active implementation and decide when to swap-off the active implementation in favor of another implementation directly at run time.

We develop a simple self-aware adaptive application using the self-aware adaptive library providing the DES encryption and decryption functionalities. This application specifies an expected performance goal over time such that the software implementation results fast enough, and the library translates this goal into an expected *heart rate*. Heartbeats make it possible to check if the current heart rate fits the expected goal, as we have seen in the previous Chapter; this enables the self-aware adaptive library to take decisions in accordance using an heuristic that avoids short-term oscillations. When the throughput (*i.e.*, heart rate) over a certain window of time drops under or excessively overcomes the expectations (and when other more subtle conditions are true) the library acts to improve performance. Since it is aware of the presence of two implementations the action consists in the switch between them.

Figure 5.6 shows an execution of the self-aware adaptive application. Even though in (t_0, t_1) the application's heart rate is dropping due to the context switches we are not observing any change in the implementation because the current heart rate is still inside the desired heart rate window delimited by m and M. In (t_1, t_2) the computed heart rate exits this window for more than Δ[1] time instants, going under the lower bound m, hence the self-aware adaptive library decides to switch from the software implementation to the hardware implementation trying to re-entering the desired heart rate window. Therefore, in (t_2, t_3) the self-aware adaptive library reconfigures the FPGA[2] and switches to the hardware implementation. In (t_3, t_4) the heart rate increases coming back within the desired heart rate window. Then, a sudden decrease of the heart rate in (t_4, t_5) proceeding in (t_5,

[1]The Δ time depends on the value B discussed in Section 4.2.

[2]The reconfiguration time R is spent only when the desired hardware implementation is not already configured on the FPGA.

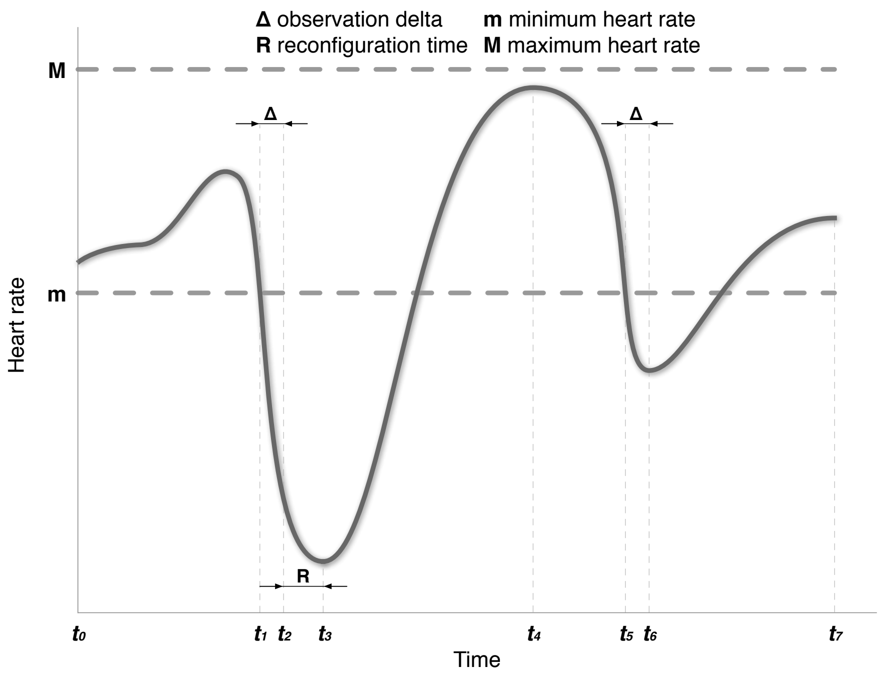


Figure 5.6: Self-aware adaptive application behavior over an execution. The interval between m and M defines the desired heart rate window

t_6) happens; this sudden drop is caused by resources contention (*i.e.,* buses and memory) due to the base based architecture. Therefore, after waiting for Δ time instants in (t_5, t_6) the software implementation is swapped in to try increasing the heart rate as it then happens in (t_6, t_7).

The execution of the self-aware adaptive application displays the validity of the proposed approach and confirms the fact that in a an unpredictable multitasking environment the performance information gained thanks a static analysis are not sufficient to guarantee an execution fulfilling a desired QoS. However, it is important to highlight that not even a self-aware adaptive computing system can really guarantee the fulfillment of a certain goal. That said, the advantage of a self-aware adaptive computing system is ability of reacting to non-favorable execution conditions (*e.g.,* resources contention) to try its best to reach the given goal.

To build our self-aware adaptive library we adopted Heartbeats[3] as the monitoring framework. This observation mechanism is the one supporting the decisioning process in performing its work. However, to truly improve performance it is necessary for the monitoring framework to be lightweight and low-overhead. To prove this we evaluated Heartbeats overhead considering the encryption of 1 to 1000 blocks with the software implementation, the result was obtained averaging the outcomes on tens of different trials.

The average impact on the execution time (*i.e.,* average overhead *read*) is a moderate, 3.52%, which seems to even become lower with a greater number of blocks (*i.e.,* see overhead *blue*) as shown in Figure 5.7 that displays the trend of the overhead. This small overhead shows once again the validity of our approach since it allows self-aware adaptive computing systems to try to reach the given performance goals with a viable overhead on the execution time.

[3] We adopted Heartbeats version 1.2 [63].

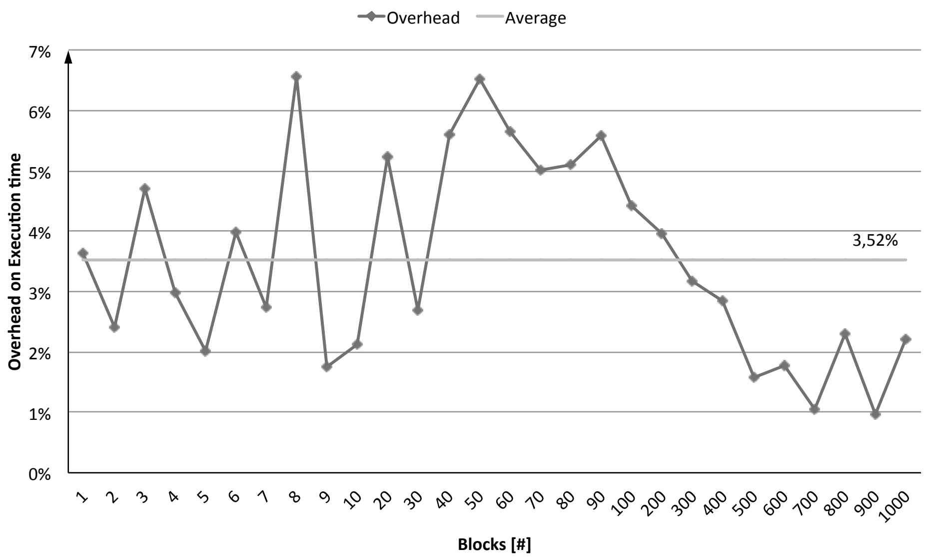


Figure 5.7: Overhead of Heartbeats over the execution time of the software implementation used to cipher 1000 blocks

5.2 Heterogeneous computing system

Since our implementation is meant to fit a wide range of computing systems, from mobile devices to cloud computing systems, we adopted not only an embedded computing system supporting dynamic reconfiguration as reported in the previous Section but also an heterogeneous computing system featuring an FPGA used as a pure-accelerator.

The testing platform is based on an x86-64 machine featuring an *Intel Core i7-870* processor [64] used as GPP, 4 GB of SDRAM DDR3-1333 of main memory, and an *NVIDIA GeForce GT 240* graphic card used as Graphics Processing Unit (GPU). The computing system is equipped with a *Xilinx University Program Virtex-5 (XUPV5)* board featuring an XC5VLX110T FPGA, connected through the Peripheral Component Interconnect (PCI) Express system bus, used as a pure-accelerator. The FPGA is configured with a PCI Express end-point, a mapper from the end-point to the local PLB bus and an

IP-Core implementing the Secure Hash Algorithm 1 (SHA1) cryptographic hash functionality which is connected to the local bus.

The computing system is equipped with *Debian GNU/Linux 64-bit* (development release) featuring the Linux kernel version 2.6.32 and GCC version 4.4.5.

A schematics of the architecture is proposed in Figure 5.8.

5.2.1 Preliminary results and validation

The following results have been obtained using an heterogenous computing system whose architecture has been described in the previous Section. A notable difference between this configuration and the System-on-Chip (SoC) adopted as embedded computing system is the fact that the GPP (*i.e.,* Intel Core i7-870 processor) is way faster and hence the software and the hardware implementations have comparable performance. Moreover, the IP-Core implementing SHA1 is interfaced to the rest of the computing system with more bridges hence there are more and more bottle necks.

This consideration is important to explain way the software and the hardware implementations have comparable static performance as reported in the proceeding of this Section.

Once again we came up with a software implementation library deployed as a DLL, an hardware implementation library which is deployed as a DLL too that wraps a PCI device driver for the Linux kernel. These two implementation libraries are accessed by means of a self-aware adaptive library which makes use of Heartbeats and the hot-swap mechanism.

Static analysis

We started analyzing the static performance of both the software and the hardware implementation libraries varying the input data size averag-

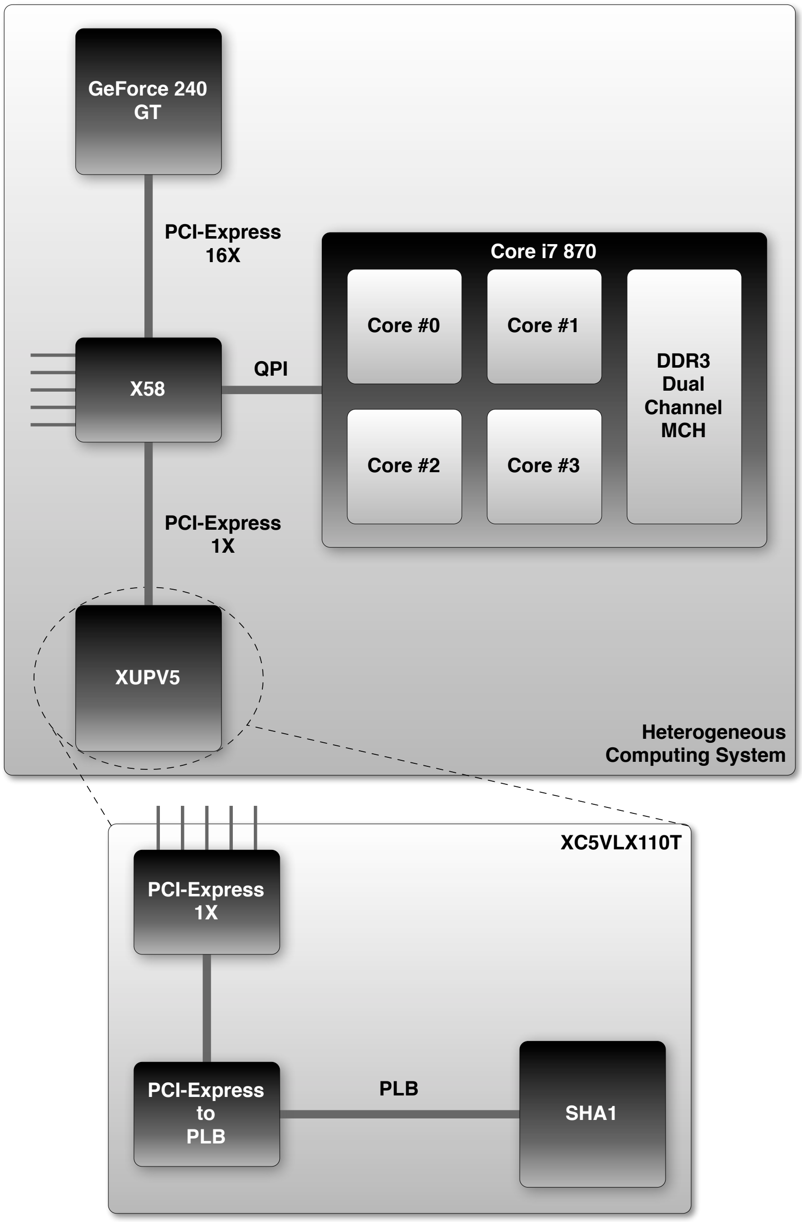


Figure 5.8: Block diagram of the heterogeneous hardware architecture made up of an Intel Core i7 870 microprocessor sided by an NVIDIA GeForce 240 GT graphic adapter and a Xilinx XUPV5 featuring an XC5VLX110T FPGA

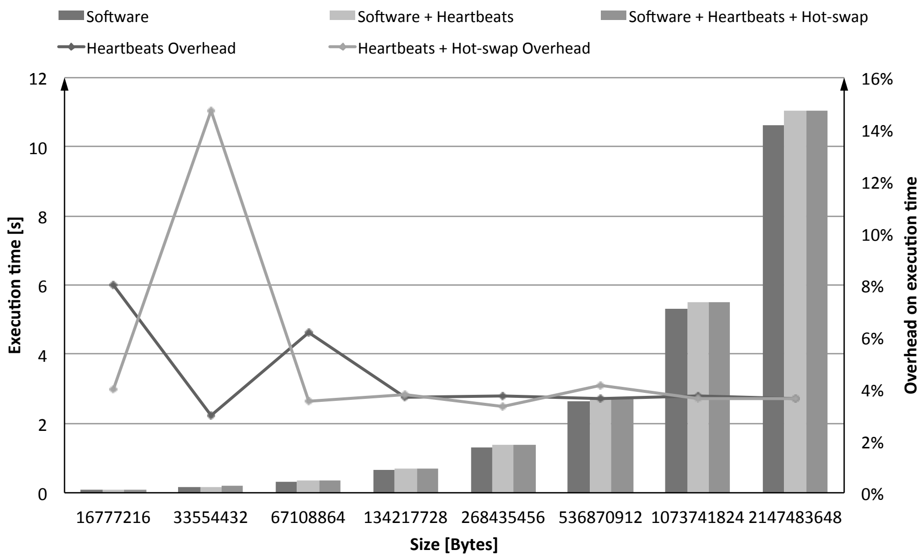


Figure 5.9: Execution times of the software implementation library to hash 64 MB to 2 GB of random generated data

ing the results on tens of different trials.

We ran both the implementation libraries as is, then we enabled the monitoring mechanism (*i.e.,* Heartbeats), and finally we enabled the hot-swap mechanism in order to measure the overhead of each self-aware adaptive sub-system with respect to the plain implementation library.

Figure 5.9 shows the results obtained running the software implementation library that highlight the moderate impact of the monitoring subsystem, 4.45% on average, and of the monitoring/decisioning/acting subsystems, 5.10% on average.

Figure 5.10 shows the results obtained running the hardware implementation library that highlight the moderate impact of the monitoring sub-system, 5.38% on average, and of the monitoring/decisioning/acting sub-systems, 6.00% on average.

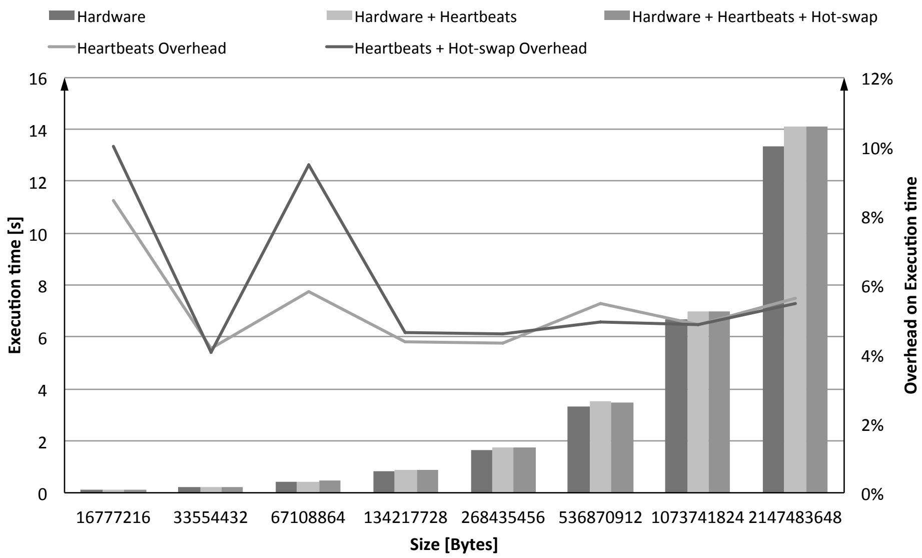


Figure 5.10: Execution times of the hardware implementation library to hash 64 MB to 2 GB of random generated data

Dynamic analysis

We already stated the static analysis of the available implementation libraries is not enough to decide which implementation best suits a given execution scenario.

Figures 5.11, 5.12, and 5.13 shows three sample execution scenarios in which a self-aware adaptive application hashes random data using SHA1 with different performance goals.

In Figure 5.11 the application requires a target heart rate between 35000 and 40000 Hz and as we can see, after an initial over shot in which the self-aware adaptive library tries to match the given goal hot-swapping among the available implementation libraries, the execution stabilizes between the desired heart rate window.

In Figure 5.12 the application requires a target heart rate between 37500 and 42500 Hz which is matched in a short amount of time; however, during

Figure 5.11: Dynamic analysis of the execution of a self-aware adaptive application hashing random data with a target heart rate between 35000 and 40000 Hz

the last portion of the execution the measured performance exceeds the maximum heart rate and hence the self-aware adaptive library starts an hot-swap cycle to re-adjust the execution.

The scenario shown in Figure 5.13 is highly disturbed. In this case the application requires a target heart rate between 40000 and 45000 Hz that is matched only at the end of the execution after a lot of hot-swapping among the available implementation libraries mainly because of the random usage of the heterogeneous computing system during the execution of the application.

The three scenarios reported above shows the full potential of a self-aware adaptive computing system which is capable to react to a changing environment and hence changing its behavior in order to match the given performance goal. In fact, the result acquired for the dynamic analysis are obtained on a computing system which is used by many different users in many different ways in order to “randomly” change the actual performance

Figure 5.12: Dynamic analysis of the execution of a self-aware adaptive application hashing random data with a target heart rate between 37500 and 42500 Hz

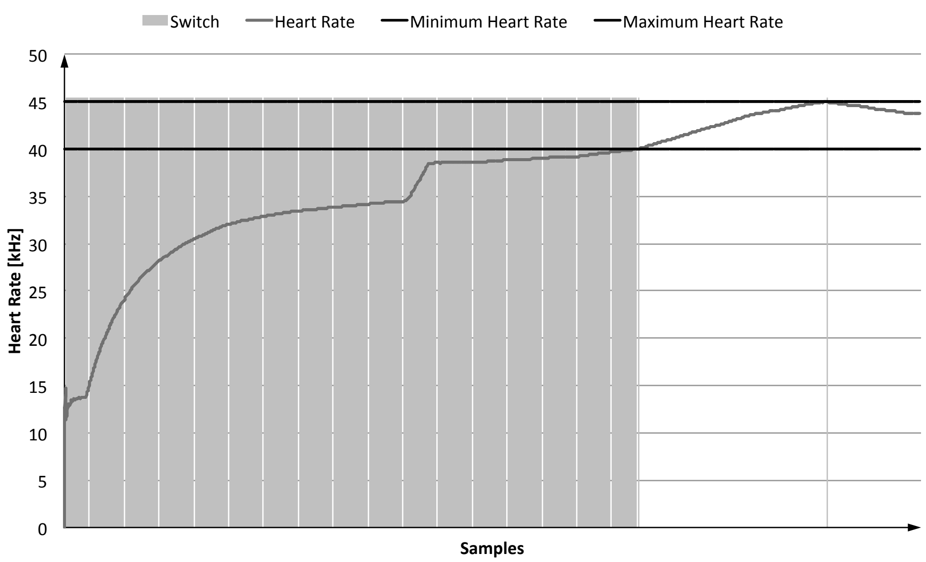


Figure 5.13: Dynamic analysis of the execution of a self-aware adaptive application hashing random data with a target heart rate between 40000 and 45000 Hz

of both the software and the hardware implementation libraries.

Chapter 6

Conclusions and Future works

Aim of this thesis work has been the proposal of an approach to build self-aware adaptive computing systems using hot-swap mechanisms to adapt their behavior in response to certain internal and environmental conditions. An implementation of a self-aware adaptive library, providing the necessary capabilities to develop applications exposing the self-aware adaptive behavior exposed in Chapter 1, has been discussed and validated. The resulting self-aware adaptive library provides applications with two different implementations of the exposed functionalities: Data Encryption Standard (DES) encryption and DES decryption. The former is a software implementation using the General Purpose Processor (GPP) of the adopted computing system while the latter is an hardware implementation studied making use of a dedicated co-processor, implemented as an Intellectual Property Core (IP-Core) for Field Programmable Gate Arrays (FPGAs) and accessed by means of device driver and the system calls' layer of the Linux kernel.

To further improve the ability of our computing system to react to internal and environmental conditions changing, we adopted a dynamically reconfigurable platform equipped with an FPGA. The operating system based on the Linux kernel installed on the computing system has been instrumented to support reconfigurable hardware devices. A device driver providing the access to a device node representing an internal reconfiguration port on the FPGA allows user mode applications to dynamically reconfigure the underlying reconfigurable hardware device and hence adapt the hardware to the needs of running applications.

A self-aware adaptive computing system follows the Observe, Decide, Act (ODA) loop, it observes, decides, and acts accordingly. For the self-aware adaptive computing systems we take in consideration, acting means adapting both the software portion and the hardware portion of the computing system to fit applications' needs by means of a series of hot-swaps between the available implementations. The decision on which implementation best suits the needs of an application is delegated to a simple yet effective heuristic that try to fulfill applications' goals avoiding excessive short-term oscillations between the available implementations. The decision mechanism is enabled by the availability of an observation mechanisms. The self-aware adaptive library make use of a lightweight and low-overhead observation mechanism based on an open source library: Heartbeats. Thanks to the capabilities of defining a performance goal over the execution of an application, updating and monitoring the execution progress by means of two simple concepts: an heartbeat and the heart rate, the library is enabled to take decisions. The definition of a self-aware adaptive process given in Chapter 3 provides a useful abstraction to understand the evolution of a self-aware adaptive process, that follows the ODA loop, running on a dynamically reconfigurable computing system - which allows hardware adaptation in addition to the software adaptation. This abstraction makes the execution of the software implementation and the hardware implementation indistinguishable for user mode applications using a self-aware adaptive library.

Experimental results shown in the static analysis make one think a statically taken decision among the software implementation, the hardware implementation, and the reconfigurable hardware implementation can be taken. However, the dynamic analysis demonstrating the behavior of the self-aware adaptive library shows the validity of the proposed approach: a static decision cannot take into account varying internal and environmental conditions. The implementation of the self-aware adaptive library correctly executes the switch between the available implementations in a transparent fashion for applications using such library. Moreover, since the underlying operating system based on the Linux kernel supports reconfigurable hardware devices, the self-aware adaptive library performs, if needed, a dynamic reconfiguration when adopting the hardware implementation. Since the benefit of the run time observation and decision mechanisms has been proved by means of the dynamic analysis in unpredictable internal and environmental conditions, we claim that the overhead introduced by those mechanisms is justified and reasonable.

The proposed implementations can be further extended in multiple ways. First of all, Heartbeats is a simple yet effective framework for performance monitoring, however, current versions somehow allow only a reduced set of performance goals (*e.g.,* blocks per second, frames per second, and so on). For example, applications using the self-aware adaptive library are allowed to choose only a performance goal in the form of: compute *number of blocks per second*. Computing systems equipped with "sensors" to evaluate the power consumption, alongside with an extended version of the monitoring framework, enable the definition of more complicated performance goals and hence the hot-swap mechanism may be fully exploited.

Second, an attractive scenario is the evaluation of the proposed implementation on heterogeneous computing systems equipped with many multi-core or many-core processors and where board based on FPGAs are used as pure-accelerators with the support for dynamic, partial, internal reconfiguration. This is a classic scenario in High-Performance Computing

(HPC) solutions such as the one from from CRAY which put together GPPs, General Purpose Graphics Processing Units (GPGPUs), and FPGAs.

An important limitation of the proposed adaptive code approach is the lack of a set of policy to manage reconfigurable hardware devices. As results shown, the dynamic reconfiguration process, which improves the set of actions a self-aware adaptive computing system can actuate to fit internal and environmental conditions, is indeed a time consuming task affecting the use of the reconfigurable hardware implementation with a huge overhead. Providing the operating system with an intermediate layer to gracefully handle the caching and preemption[1] of IP-Cores on reconfigurable hardware devices will surely benefit self-aware adaptive computing system equipped with reconfigurable hardware devices. Such intermediated layer has already been implemented and validated in [19].

Chapter 3 and Chapter 4 describe an additional approach to build self-aware adaptive computing systems. The key idea is to provide support for hot-swap among available implementations of the same functionality or components within a mainstream and widespread operating system kernel: the Linux kernel. The benefit to adopt a massively diffused solution consists in the potential to provide a huge number of applications with a technology to enable a self-aware adaptive behavior: an hot-swap mechanism. Even though the Linux kernel is indeed flexible and extendible, it was not designed with self-aware adaptive concepts in mind, therefore, the implementation of an hot-swap mechanism first requires the introduction of some keys enabling technologies. The first enabling technology refers to the switchable unite: the Loadable kernel modules (LKMs) which must be designed and developed following the stacked approach proposed in this thesis work. One of the future works of this thesis consists in completing

[1] IP-Cores preemption is useful to interrupt the execution of an hardware implementation in order to activate de-fragmentation techniques to bolster the use of the available reconfigurable area.

the implementation of the hot-swap mechanism for LKMs within the Linux kernel.

Appendix A

Interposition in K42

The *interposition mechanisms* is a specialization of the interposition process described in Chapter 2.3.4. The ability to interpose an object between the client and an existing object adds additional functionalities around all function calls of the existing object. This mechanism basically needs two distinct pieces: a *generic interposer* and a *wrapper object* [23, 34].

The generic interposer - simply interposer from now on - performs similar functions the mediator performs in the hot-swap mechanism. The first step by the interposer is to replace the original object references in the both the Global Translation Table (GTT) entry and the Local Translation Tables (LTTs) entries with references to itself following the same schema reported in Chapter 2.3.4. All subsequent calls to the original object go through the interposer, requiring that it provides transparent call forwarding to the component behind it. The call forwarding mechanisms is the same previously discussed.

Once the interposer replaces the original object in the LTTs, all calls go through the interposer's virtual function table whose methods are overloaded with a single interposition method, forcing all clients' calls through this method. This interposition method handles calls to the wrapper object's methods as well as calls to the appropriate method of the original

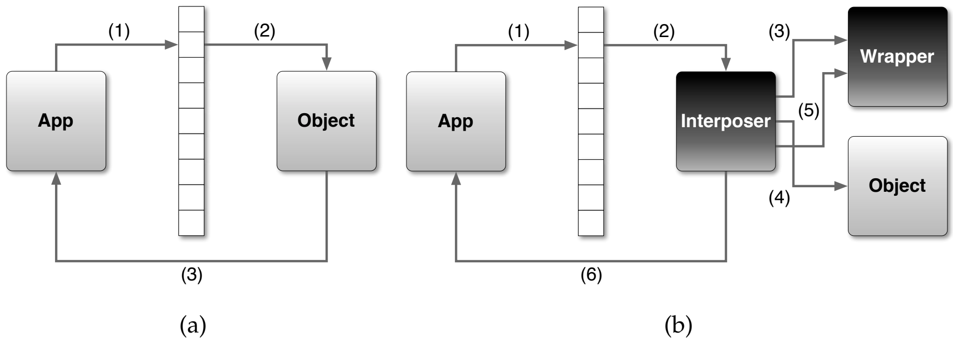


Figure A.1: Interposition process: (a) prior phase, the object is working in normal conditions; (b) post phase or interposed state, a generic interposer and a wrapper object are instantiated. The interposer intercepts all calls directed to the object and wraps them using `precall` (3) and `postcall` (5) provided by the wrapper

object.

The wrapper object - wrapper from now on - is a standard C++ object with two calls: `precall` and `postcall`. The former is called before the original object's methods while the latter is called after. In these calls, a wrapper can maintain state about each calls that is in flight, collect statistical information, modify call's parameters and return values, and so on.

Figure A.1 shows the interposition process divided in its steps. Figure A.1a is the initial state of both the client and the original object. To perform the interposition, instances of the generic interposer and the wrapper are created. The interposer keeps a reference to the original object and replaces its entries in the LTTs with references to itself. This is shown in Figure A.1b. At this point, all clients' calls to the original object are handled by the the interposer. When the interposer receives a call, it calls the wrapper's `precall`, calls the original object's method, calls the wrapper's `postcall`, and finally, returns to the caller.

This partitioned approach, the use of an interposer alongside with a specialized wrapper object exporting a precise interface, allows to implement a single generic interposer and a variety of wrapper objects tailored on the specific component they are meant to extend.

To detach an interposed wrapper, the corresponding interposer simply replaces its LTTs entries with pointers to the original object.

Appendix B

Dynamic updates in K42

The K42 operating system is structured in such a way each resource is managed by an object instance. Therefore, if the system has two resources of the same type there must be two different instances of the same object to manage them. The Hot-swap mechanism enables the switch of a Clustered Object (CO) instance, and was designed to enable adaptability. This mechanism was further extended to support dynamic update [65, 39, 66, 67] enabling the switch of every active instance of a CO.

Before the dynamic update mechanism introduction COs instances were created through calls to statically bound `Create` methods. Each `Create` method is bound at compile time thus cannot be redirected to an updated implementation. To address these problem the factory design pattern was adopted. The responsibility for creating and tracking COs is now placed with a factory object.

Every factory object - factory from now on - is a CO itself and is implemented in a distributed fashion. To perform a dynamic update of a CO, the following steps, also illustrated in Figure B.1, are taken:

1. the new factory, which handles the new implementation of the target CO is instantiated, see Figure B.1b;

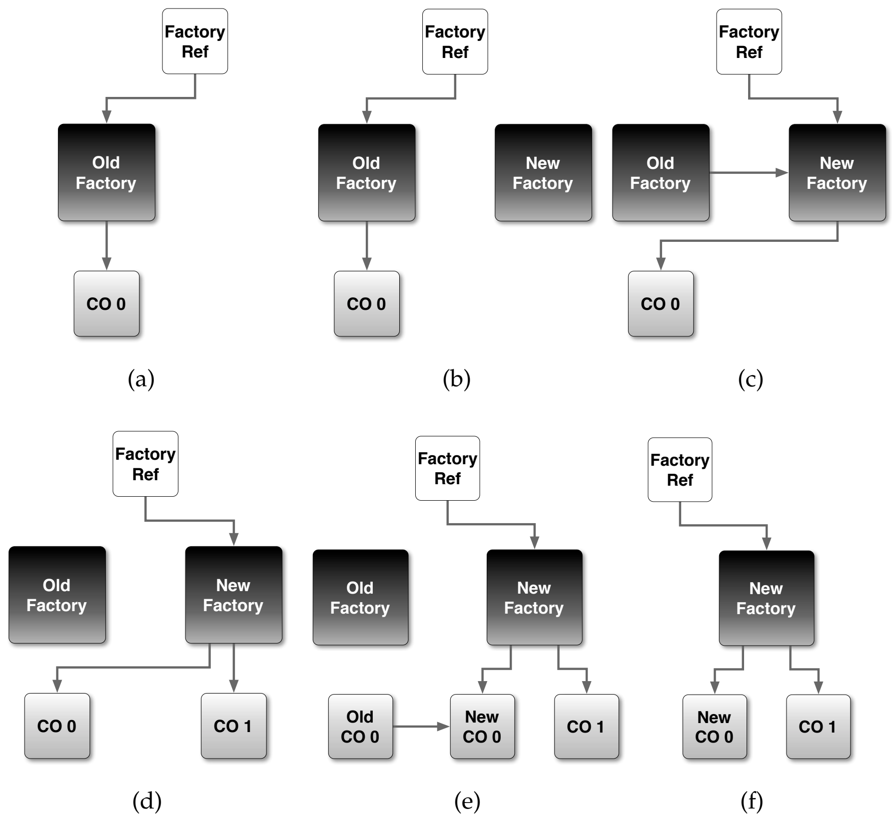


Figure B.1: Dynamic update process: (a) prior phase, the old factory hold the reference to a CO; (b) the new factory is instantiated; (c) the new factory is set as default factory for CO set and the old factory is hot-swapped with the new factory; (d) the new factory is already working and a CO - with the new implementation - is instantiated; (e) the old CO is hot-swapped with a new CO featuring the new implementation; (f) both the old factory and the old COs are de-instantiated

2. the factory ref is switched to the new factory thanks to the hot-swap mechanism. During this process the new factory receives the list of instances that was being maintained by the old factory, see Figure B.1c;

3. the new instances of the target CO are handled by the new factory, therefore, they are instances of the new implementation, see Figure B.1d;

4. the old instances of the target CO are updated by the new factory traversing the list it received from the old factory, and for every entry it creates a new instance of the target CO and performs a switch between the old instance and the new one. This step proceeds in parallel across all processors where the old factory was in use, see Figure B.1e;

5. the old factory is finally destroyed as shown in Figure B.1f.

Bibliography

[1] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[2] Petre Dini, Wolfgang Gentzsch, Mark Potts, Alexander Clemm, Mazin S. Yousif, and Andreas Polze. Internet, GRID, Self-Adaptability and Beyond: Are We Ready? In *Proceedings of the 15th International Workshop on Database and Expert Systems Applications, DEXA 2004*, pages 782–788, Zaragoza, Spain, August 30th-September 3rd 2004. IEEE Computer Society.

[3] IBM Corporation. An Architectural Blueprint for Autonomic Computing. `http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf`, June 2006.

[4] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems, TAAS*, 4(2), 2009.

[5] Gerald Estrin. Organization of Computer Systems-The Fixed Plus Variable Structure Computer. In *Proceedings of the Western Joint Computer Conference*, pages 33–40, New York, NY, USA, 1960.

[6] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. PetaBricks: A Language

and Compiler for Algorithmic Choice. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, pages 38–49, Dublin, Ireland, June 15^{th}-21^{st} 2009. ACM.

[7] Cy P. Chan, Jason Ansel, Yee Lok Wong, Saman P. Amarasinghe, and Alan Edelman. Autotuning Multigrid with PetaBricks. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009*, Portland, Oregon, USA, November 14^{th}-20^{th} 2009. ACM.

[8] Johnathan Eastep, Harshad Kasture, and Anant Agarwal. The Organic Template Library: A Parallel Runtime-Adaptive C++ Library. Technical report, 4^{th} CSAIL (MIT) Student Workshop, Gloucester, MA, US, September 22^{th} 2008.

[9] Anant Agarwal, Jason Miller, Jonathan Eastep, David Wentziaff, and Harshad Kasture. Self-Aware Computing. Technical report, MIT+DARPA, June 2009.

[10] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. Standard Templates Adaptive Parallel Library (STAPL). In David R. O'Hallaron, editor, *Proceedings of the 4^{th} International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, LCR '98*, volume 1511 of *Lecture Notes in Computer Science*, pages 402–409, Pittsburgh, PA, USA, May 28^{th}-30^{th} 1998. Springer.

[11] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Program-*

ming, PPOPP 2005, pages 277–288, Chicago, IL, USA, June 15$^{\text{th}}$-17$^{\text{th}}$ 2005. ACM.

[12] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A Self-Reconfiguring Platform. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Proceedings of the 13$^{\text{th}}$ International Conference on Field Programmable Logic and Applications, FPL 2003*, volume 2778 of *Lecture Notes in Computer Science*, pages 565–574, Lisbon, Portugal, September 2003. Springer.

[13] Grant B. Wigley and David A. Kearney. The First Real Operating System for Reconfigurable Computers. In *Proceedings of the 6$^{\text{th}}$ Australasian Computer Systems Architecture Conference, ACSAC 2001*, pages 130–137, Gold Coast, Queensland, Australia, January 29$^{\text{th}}$-30$^{\text{th}}$ 2001. IEEE Computer Society.

[14] Ivan Gonzalez, Francisco J. Gomez-Arribas, and Sergio López-Buedo. Hardware-Accelerated SSH on Self-Reconfigurable Systems. In Gordon J. Brebner, Samarjit Chakraborty, and Weng-Fai Wong, editors, *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, FPT 2005*, pages 289–290, Singapore, December 11$^{\text{th}}$-14$^{\text{th}}$ 2005. IEEE.

[15] Embedded Linux Microcontroller Project. `http://www.uclinux.org/`.

[16] John W. Williams and Neil W. Bergmann. Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip. In Toomas P. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA'04*, pages 163–169, Las Vegas, Nevada, USA, June 21$^{\text{st}}$-24$^{\text{th}}$ 2004. CSREA Press.

[17] Alberto Donato, Fabrizio Ferrandi, Massimo Redaelli, Marco D. Santambrogio, and Donatella Sciuto. Caronte: A methodology for the Implementation of Partially dynamically Self-Reconfiguring Systems on FPGA Platforms. In Ricardo Augusto da Luz Reis, Adam Osseiran, and Hans-Jörg Pfleiderer, editors, *VLSI-SoC: From Systems To Silicon, Proceedings of IFIP 13th International Conference on Very Large Scale Integration of System on Chip, VLSI-SoC 2005*, volume 240 of *IFIP*, pages 87–109, Perth, Australia, October 17th-19th 2007. Springer.

[18] Xilinx, Inc. OPB HWICAP: Product Specification. `http://www.xilinx.com/support/documentation/ip_documentation/opb_hwicap.pdf`, March 15th 2004.

[19] Marco D. Santambrogio, Vincenzo Rana, and Donatella Sciuto. Operating system support for online partial dynamic reconfiguration management. In *Proceedings of the 18th International Conference on Field Programmable Logic and Applications, FPL 2008*, pages 455–458, Heidelberg, Germany, September 8th-10th 2008. IEEE.

[20] Hayden Kwok-Hay So, Artem Tkachenko, and Robert W. Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In Reinaldo A. Bergamaschi and Kiyoung Choi, editors, *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2006*, pages 259–264, Seoul, Korea, October 22nd-25th 2006. ACM.

[21] Hayden Kwok-Hay So and Robert W. Brodersen. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, CA, USA, July 2007.

[22] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc Auslander, David Edel-

sohn, Ben Gamsa, Gregory R. Ganger, Paul Mckenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 2003.

[23] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, and Jimi Xenidis. System Support for Online Reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference, USENIX 2003*, pages 141–154, San Antonio, Texas, USA, June 9th-14th 2003. USENIX.

[24] Mesaac Makpangou, Yvon Gourhant, and Jean pierre Le Narzul. Fragmented Objects for Distributed Abstractions. In *Proceedings of the Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1992.

[25] Philip Homburg, Leendert Van Doorn, Maarten Van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge. An Object Model for Flexible Distributed Systems. In *Proceedings of the First ASCI Annual Conference*, pages 69–78, 1995.

[26] Jonathan Appavoo, Marc A. Auslander, Maria A. Butrico, Dilma Da Silva, Orran Krieger, Mark F. Mergen, Michal Ostrowski, Bryan S. Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–441, 2005.

[27] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares.

Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems, TOCS*, 25(3), 2007.

[28] Jochen Liedtke. On micro-kernel Construction. *SIGOPS Operating System Review*, 29(5):237–250, December 1995.

[29] Dilma Da Silva, Livio B. Soares, and Orran Krieger. KFS: Exploring Flexibility in File System Design. `http://www.research.ibm.com/K42/papers/wsoBrazil2004.pdf`, August 2004.

[30] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc A. Auslander, David Edelsohn, Benjamin Gamsa, Gregory R. Ganger, Paul E. McKenney, Michal Ostrowski, Bryan S. Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[31] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Scheduling in K42. `http://www.research.ibm.com/K42/white-papers/Scheduling.pdf`, August 2002.

[32] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Memory Management in K42. `http://www.research.ibm.com/K42/white-papers/MemoryMgmt.pdf`, August 2002.

[33] Benjamin Gamsa, Orran Krieger, Eric W. Parsons, and Michael Stumm. Performance Issues for Multiprocessor Operating Systems. Technical report, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, November 1995.

[34] Jonathan Appavoo. *Clustered Objects*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 2005.

[35] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 3^{rd} edition, November 2005.

[36] Marco D. Santambrogio, Henry Hoffmann, Jonathan Eastep, and Anant Agarwal. Enabling Technologies For Self-Aware Adaptive Systems. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2010*, Anaheim Convention Center, Anaheim, CA, USA, June 15^{th}-18^{th} 2010.

[37] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceedings of the 7^{th} IEEE/ACM International Conference on Autonomic Computing and Communications*, pages 215–224, Washington, DC, USA, June 7^{th}-11^{th} 2010. IEEE/ACM.

[38] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Controlling software applications via resource allocation within the Heartbeats framework. In *Proceedings of the 49^{th} 49th IEEE Conference on Decision and Control*, Atlanta, Georgia, USA, December 15^{th}-17^{th} 2010. IEEE.

[39] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing Dynamic Update in an Operating System. In *Proceedings of the 2005 USENIX Annual Technical Conference, USENIX 2005*, pages 279–291, Anaheim, CA, USA, April 10^{th}-15^{th} 2005. USENIX.

[40] Filippo Sironi, Marco Triverio, Henry Hoffmann, Martina Maggio, and Marco D. Santambrogio. Self-Aware Adaptation in FPGA-based Sys-

tems. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications, FPL 2010*, Milano, Italy, August 31st-September 2nd 2010.

[41] U.S. DEPARTMENT OF COMMERCE National Institute of Standards and Technology. Data Encryption Standard (DES). FIPS PUB 46-3, Federal Information Processing Standards Publication: `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`, October 25th 1999.

[42] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats for Software Performance and Health. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010*, pages 347–348, Bangalore, India, January 9th-14th 2010. ACM.

[43] Erich Gamma, Ralph Johnson, Richard Helm, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 10th 1994.

[44] U.S. DEPARTMENT OF COMMERCE National Institute of Standards and Technology. Secure Hash Standard (SHS). FIPS PUB 180-3, Federal Information Processing Standards Publication: `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`, October 2008.

[45] The Internet Society. US Secure Hash Algorithm 1 (SHA1). Network Working Group, Request for Comments (RFC): 3174: `http://www.rfc-editor.org/rfc/rfc3174.txt`, September 2001.

[46] M. Tim Jones. Anatomy of Linux loadable kernel modules. http://download.boulder.ibm.com/ibmdl/pub/software/dw/linux/l-lkm/l-lkm-pdf.pdf, July 16th 2008.

[47] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, EuroSys 2009*, pages 187–198, Nuremberg, Germany, April 1st-3rd 2009. ACM.

[48] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA, 3rd edition, February 2005.

[49] Xilinx, Inc. PowerPC 405 Processor Block: Reference Guide. http://www.xilinx.com/support/documentation/user_guides/ug018.pdf, January 11th 2010.

[50] Xilinx, Inc. Virtex-II Pro™ and Virtex-II Pro™ X Platform FPGAs: Product Specification. http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf, November 5th 2007.

[51] Xilinx, Inc. Multi-Port Memory Controller (MPMC): Product Specification. http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf, December 2nd 2009.

[52] The Linux Kernel. http://www.kernel.org/.

[53] BusyBox: The Swiss Army Knife of Embedded Linux. http://www.busybox.net/.

[54] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[55] uClibc: A C library for embedded Linux. http://www.uclibc.org/.

[56] Xilinx, Inc. XPS SYSACE (System ACE) Interface Controller Product Specification. http://www.xilinx.com/support/documentation/ip_documentation/xps_sysace.pdf, December 2nd 2009.

[57] Xilinx, Inc. XPS UART Lite: Product Specification. http://japan.xilinx.com/support/documentation/ip_documentation/xps_uartlite.pdf, April 20th 2007.

[58] C. Stephen Carr. RFC15: Network Subsystem for Time Sharing Hosts. http://tools.ietf.org/html/rfc15, September 25th 1969.

[59] J. Postel and J. Reynolds. RFC854: Telnet Protocol Specification. http://tools.ietf.org/html/rfc854, May 1983.

[60] Xilinx, Inc. XPS Ethernet Lite Media Access Controller: Product Specification. http://china.xilinx.com/support/documentation/ip_documentation/xps_ethernetlite.pdf, April 19th 2010.

[61] OpenCores.org. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. http://www.opencores.org/downloads/wbspec_b3.pdf, September 7th 2002.

[62] IBM Corporation. The CoreConnect Bus Architecture. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569910050C0FB/$file/crcon_wp.pdf.

[63] CSAIL (MIT). Application Heartbeats Website. http://groups.csail.mit.edu/carbon/heartbeats/Welcome.html.

[64] Intel Corporation. Intel Core i7-870 Processor. `http://ark.intel.com/Product.aspx?id=41315&processor=i7-870&spec-codes=SLBJG`.

[65] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Improving Operating System Availability With Dynamic Update. In *Proceedings of the First Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, pages 21–27, 2004.

[66] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Module Hot-Swapping for Dynamic Update and Reconfiguration in K42. In *Proceedings of the 6$^{\text{th}}$ Linux.Conf.Au, LCA*, Canberra, Australia, April 2005.

[67] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the 2007 USENIX Annual Technical Conference, USENIX 2007*, pages 337–350, Santa Clara, CA, USA, June 17$^{\text{th}}$-22$^{\text{nd}}$ 2007. USENIX.

Vita

Filippo Sironi

Education

- Master of Science in Computer Science at University of Illinois at Chicago (final grade: 3.85/4.00).
- Laurea (equivalent to Bachelor of Science) in Ingegneria Informatica at Politecnico di Milano (final grade: 105/110).
- Diploma di Istruzione Superiore (equivalent to High School Diploma) in Informatica e Telecomunicazioni at Istituto di Istruzione Superiore Jean Monnet di Mariano C.se (final grade: 100/100).

Publications

1. Filippo Sironi, Marco Triverio, Henry Hoffmann, Martina Maggio, and Marco D. Santambrogio. Self-Aware Adaptation in FPGA-based Systems. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications, FPL 2010*. Milano, Italy, August 31st-September 2nd 2010.

Professional experiences

- IT Consultant for DIPROS Italia s.r.l., Via Prealpi, 13, 20034 Giussano, Monza e Brianza, Italy (Reference: Renato Sironi).

- Tirocinio di Formazione ed Orientamento (equivalent to Stage) in Industrial Automation at S.L.A.I s.n.c., 20050 Villa Raverio di Besana Brianza, Monza e Brianza, Italy (Reference: Ferdinando Dell'Oro).