Co-managing Software and Hardware Modules through the Juggle Middleware

Jan S. Rellermeyer, Ramon Küpfer

Co-Managing Software and Hardware Modules through the Juggle Middleware

Jan S. Rellermeyer[1]* and Ramon Küpfer[2]

[1] IBM Austin Research Laboratory,
Austin, TX
rellermeyer@us.ibm.com
[2] Department of Computer Science, ETH Zurich,
Zurich, Switzerland
rkuepfer@student.ethz.ch

Abstract. Reprogrammable hardware like Field-Programmable Gate Arrays (FPGAs) is becoming increasingly powerful and affordable. Modern FPGA chips can be reprogrammed at runtime and with low latency which makes them attractive to be used as a dynamic resource in systems. For instance, on mobile devices FPGAs can help to accelerate the performance of critical tasks and at the same time increase the energy-efficiency of the device. The integration of FPGA resources into commodity software, however, is a highly involved task. On the one hand, there is an impedance mismatch between the hardware description languages in which FPGAs are programmed and the high-level languages in which many mobile applications are nowadays developed. On the other hand, the FPGA is a limited and shared resource and as such requires explicit resource management. In this paper, we present the Juggle middleware which leverages the ideas of modularity and service-orientation to facilitate a seamless exchange of hardware and software implementations at runtime. Juggle is built around the well-established OSGi standard for software modules in Java and extends it with support for services implemented in reprogrammable hardware, thereby leveraging the same level of management for both worlds. We show that hardware-accelerated services implemented with Juggle can help to increase the performance of applications and reduce power consumption on mobile devices without requiring any changes to existing program code.

Keywords: OSGi, FPGA, Hardware Acceleration

1 Introduction

The increasing degree of dynamism in modern systems design and the resulting need for more flexible software becomes particularly apparent in mobile devices. Traditionally, mobile devices implement much of their performance-critical

* The work was done while the author was at ETH Zurich, Switzerland. Part of this work was funded by the Swiss National Science Foundation SNF ProDoc program and by the Microsoft ICES initiative.

functionality in *ASICs*, application-specific integrated circuits with a fixed implementation. Once manufactured and implemented in a mobile device the ASIC cannot be changed, e.g., for extending the functionality of the device or for applying critical updates. *FPGAs* (field-programmable gate array), in contrast, are known for their reconfiguration support and their ability to change the implementation of their functionality. With the technological advances in FPGAs, partially reconfigurable chips have been developed which can alter parts of their fabric at runtime.

An example of a mobile device that already makes use of reprogrammable hardware for on-demand acceleration is the Sony Playstation Portable and its Virtual Mobile Engine [15]. The main challenge, however, is the integration of FPGAs into applications. Whereas FPGAs are programmed in low-level hardware description languages like VHDL [6] or Verilog [7], the application software on mobile devices is often developed in high-productivity languages like Java, JavaScript, or Objective C. Bridging the gap between these two worlds is far away from being trivial, especially designing the communication interfaces between applications and FPGAs and effectively managing both the reprogrammable hardware and the software side of applications. Furthermore, in practice the design of the device mandates specific patterns of interaction.

Mobile phones, for instance, are naturally constrained in the way humans can interact with them due to their form factor. For a systems design, however, this means that most of the time a mobile phone is used for exactly one interactive (foreground) task whereas the remaining tasks are running in the background and have a lower priority. For example, when the user receives a phone call, the web browser functionality of the device becomes secondary. Ideally, a system could exploit this interaction pattern by using the reconfigurable hardware for always accelerating the interactive task. In the example of the phone call, this would be the audio encoding and decoding. Since the hardware is reconfigurable, the system can keep the invariant of accelerating the most critical interactive task even when the user switches from one application to another.

Implementing such systems requires the developer to overcome the impedance mismatch between hardware and software and an active handling of the inherent dynamism of the problem, which typically results in ad-hoc solutions. The contribution of this paper is the approach of creating an equivalence between software and hardware functionality by treating both as modules—running on and being managed by a common middleware platform. In order to do so, several concrete challenges need be solved:

- handling and managing both software and *hardware modules* where the latter are the different binary images (*bitstreams*) used to reconfigure the FPGA hardware.
- the ability to substitute one implementation of a functionality with another, e.g., a software module with an accelerating hardware module and back.
- making decisions when to do substitutions, given that the hardware resource is constrained so that typically not all tasks can run in hardware at the same time.

We provide a solution to these challenges with our implementation, *Juggle*, which takes advantage of the widely-used OSGi [12] standard for dealing with the life-cycle of software modules and extends it with support for functionality implemented in reconfigurable hardware. In contrast to approaches like, e.g., Liquid Metal [2], Juggle does not attempt to apply a unified design strategy for hardware and software in the small but instead focuses on the composition and interaction in the large. Juggle then takes care of the co-existence of software and hardware implementations and provides a unified model of communication through loosely-coupled services. Based on application-dependent policies, the system can thus dynamically switch between hardware and software implementation to accelerate most critical tasks without interrupting the system. As we show in the paper, the latency for switching is low enough to allow for dynamic replacement while the achieved acceleration for the evaluated use case of an encryption service reaches a factor of 20. The amount of energy consumed can be reduced by more than 97% compared to the same encryption done in Java and 59% when comparing to an implementation in C.

2 Background

Reprogrammable hardware like Field-Programmable Gate Arrays (FPGAs) is increasingly becoming powerful and affordable which makes them attractive to be used as a dynamic resource in systems. The following sections provide background information about FPGAs and their reconfiguration and discusses OSGi for managing software modules in Java.

2.1 Field-Programmable Gate Arrays (FPGAs)

Traditional integrated circuits are the result of a manufacturing process; once they are manufactured they cannot be altered any more. FPGAs, in contrast, are integrated circuits with the ability to be reconfigured after manufacturing either by the designer itself or the customer. This advantage especially comes into effect when the implemented functionality undergoes changes—one-time changes as in product line customization or continuous changes as through periodic upgrades.

Internally, an FPGA is structured into three main parts: a set of configuration logic blocks (CLB), a programmable interconnection network between the blocks, and a set of input and output cells around the device. The actual implementation of a configuration logic block (or basic block) can vary and depends on the concrete FPGA chip used.

Juggle has been prototyped on a Virtex-II Pro chip, which consists of groups of four *slices*, each containing two actual basic blocks. A basic block consists of a lookup table (LUT) with 4 inputs and an output, a set of multiplexers, arithmetic logic and a storage element. The LUT is a group of memory cells which contain all the possible results of a given function for a given set of input values. It can therefore implement arbitrary mappings between input and output ports.

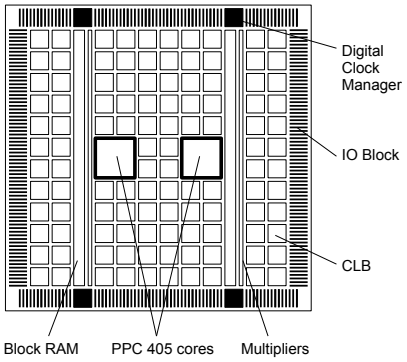

Fig. 1. Virtex-II Pro chip layout

Altering the content of a LUT through a configuration consequently changes the behavior of the basic block.

The FPGA chip is a 2-dimensional array with the CLB as the smallest element. The precise layout of the FPGA structure such as the arrangement of the logic blocks and the interconnection paradigm of the logic blocks is vendor-dependent. Figure 1 shows a schematic picture of the Virtex-II Pro FPGA. The interconnect fabric is generally a network of vertical and horizontal wires arranged in a mesh topology. At the intersection points are programmable multiplexers facilitating the routing inside the FPGA fabric. Around the periphery of the FPGA chip are the I/O components used for communication with off-chip components. Those I/O components are programmable just like the CLBs and can use as input, output, or bidirectional gates.

The programming of an FPGA typically starts with a design of the required functionality in a hardware description language like VHDL or Verilog. This design can be considered as an abstract description without taking a particular technology into account. It operates on the register transfer level (RTL), a behavioral description in terms of signal flows between hardware registers. The mapping to a concrete technology is the task of an electronic design automation (EDA) tool, which synthesizes a *netlist* from the HDL code. This netlist now describes concrete gates and could be implemented in actual hardware. However, netlists only describe instances of gates, ports, and the wiring in between but not a concrete topology. It is the task of a place-and-route tool to create an instance of the template-like netlist which resembles a concrete layout of a digital circuit. In the case of an FPGA, it represents a configuration of the FPGA fabric that implements the designed functionality.

2.2 Reconfiguration

The data to configure an FPGA is called a *bitstream*. Bitstreams can be downloaded to the device via several configuration ports, e.g., a JTAG interface or a USB cable. This has the effect that the LUTs and the routing fabric of the FPGA is changed and hence the behavior is altered. Full reconfiguration—the process of rewriting the complete design of the FPGA chip—requires a reconfiguration-time linear to the size of the bitstream to write. In addition, the entire chip is inoperable during the reconfiguration and running processes are interrupted.

Many chips therefore support the rewriting of only a part of the fabric. Specific regions on the chip are marked as reconfigurable and can at runtime be reconfigures through a *partial bitstream* while the remainder of the fabric can continue to operate. In order to support partial reconfiguration in a design, the FPGA fabric is partitioned in into a static region holding the functionality critical for the running of the system—e.g., the bus systems—and one or more regions that are partially reconfigurable (*PRRs*).

Functional tasks (reconfigurable modules or *PRMs*) can be mapped into individual PRRs (space multiplexing). If the tasks are mutually independent, they can also be mapped into the same PRMs (time multiplexing) to reduce the required FPGA real estate but at the same time introducing reconfiguration latency into the system. However, partial reconfiguration does not automatically mean that the board continues operation during reconfiguration. Depending on the hardware it can be the case that the reconfiguration requires the board to be in an inactive state. The ability of a chip to be reconfigured during runtime without interruption of the system is called dynamic partial reconfiguration.

One example of such a chip is the Xilinx Virtex-II Pro which was used to prototype Juggle. A complete discussion of the prototype system follows in Section 5. Partial reconfiguration requires the designer of a system to explicitly mark areas of the chip as reconfigurable. The place-and-route software has to take care that no signal lines are crossing these areas so that dynamic reconfiguration becomes possible. Otherwise the static part of the chip could encounter malfunctions during reconfiguration or even be short-circuit.

2.3 OSGi

In the domain of software modules, OSGi is a widely used middleware system for running and managing dynamic modules in Java. Historically, OSGi has its origins in embedded systems and mobile devices. Due to its flexibility and agility, it has recently been widely adopted in the latest generation of Java enterprise application servers. OSGi describes a runtime system that sits atop the Java virtual machine and provides primitives for controlling the life-cycle of software modules. At runtime, new modules can be installed and modules no longer needed can be completely removed from the system. Furthermore, OSGi supports consistent updates of modules. The unit of modularity in OSGi is the *Bundle*. From a technical perspective, a bundle is nothing but an ordinary JAR file—a compressed filesystem with a manifest as commonly used in Java—but

enriched with additional meta-data. Most importantly, bundles have to declare their dependencies explicitly. The default case in OSGi is that bundles do not share any code but run in complete isolation. Sharing is possible when corresponding Java packages contained in a bundle are declared to get exported and consequently are imported by another bundle. This indeed creates a tight coupling between bundles since the importing bundle cannot be resolved without an exporter already installed.

Orthogonal to the module layer, OSGi provides applications with a service layer to facilitate a loose coupling among components. Every bundle can register any Java object with the runtime system under one or more service interfaces. The OSGi runtime maintains a central service registry through which bundles can search for services. Consuming bundles are typically only tightly coupled to the service interface but no longer to the service implementation with all its transitive dependencies. When a service is acquired by a bundle, it gets the actual Java service object so that no further overhead other than the initial interaction with the runtime can be observed.

An important difference between traditional application design and the OSGi world is the handling of dynamism. Whereas usually software is assumed to be a static and unchangeable unit, in OSGi a module should never make assumptions about the permanent availability of any other module or service. An operator could at any time unload a module or stop it, which causes the removal of all registered services. Hence, OSGi bundles typically register listeners to get informed about changes in the topology and react accordingly. As a further consequence, OSGi bundles are usually written with a high degree of locality so that exchanging one service implementation with another can often be done seamlessly.

3 Management and Substitution of Modules

Introducing an FPGA into a mobile or embedded system enables applications to implement parts of their performance-critical functionality in hardware. These hardware modules are physically handled as bitstream files. Reprogramming the device requires the writing of a bitstream to a reconfiguration device embedded into the system. The first step in making FPGAs easier to use in applications is to provide management of the bitstreams of the same quality as for software modules. In a system like OSGi, this gives both the application itself and an external operator the possibility to explicitly control the composition of an application and the life-cycle of the individual components. However, this alone does not solve the integration problem. Whereas software modules can be seamlessly used in the programming languages (e.g., in OSGi through package imports), FPGAs constitute hardware components and have much more low-level communication interfaces like memory-mapped I/O ports, registers, or interrupts. In order to preserve the full flexibility of modularity, the interfaces between a software and the corresponding hardware module have the be uniform. In prac-

tice, this means that the representation of the FPGA requires the co-design of a *device driver* in software which embeds it into the host programming language.

3.1 Hardware-Accelerated Services

A hardware module in the first place consists of a partial bitstream designed to configure the core functionality of the service into a partial configuration region of the FPGA. The bitstream is embedded into the OSGi bundle as a file. Once this bitstream is applied to the FPGA, the hardware is ready to be used but still not accessible from Java. The virtual machine approach prevents Java code from accessing the underlying physical machine. Hence, the device driver is typically coded in C and makes use of the Java Native Interface (JNI) to bridge between Java and the hardware. From the point of view of the Java OSGi application, the device driver is represented through a Java class in which all critical methods map to JNI native code methods.

When loaded, the JNI code initializes by mapping the hardware addresses into the virtual memory of the JVM process as well as registering handlers for interrupts or initializing DMA. For each service method, there is a piece of code turning the service call into one or more interactions with the FPGA. Usually, this involves a mangling of the arguments and selective stores and loads of portions of the arguments into memory, waiting for a result to become available and then preparing the return value for caller. Even though writing the driver is still a challenge that requires a skilled programmer, most of this can be done in a more declarative way that takes full advantage of having a clear specification of the hardware interface on the one hand and the high-level service interface on the other. For instance, the driver code could be generated from the domain-specific language (like, e.g., in Devil [9]).

The pair of driver and bitstream is the foundation for the hardware service and dual to the Java software service implementation of the same service. When the hardware service implements the same interface it can replace the latter on demand. There might, however, be cases in which a certain consumer of a service should get accelerated by a hardware implementation while others should continue to run against a software implementation. Such a pattern of interaction is far away from being trivial to implement in OSGi since in general the application chooses the service and not the service the application.

In order to still support for such use cases we introduce *Co-Modules*, which are modules providing both a software and hardware implementation of the service at the same time. Such modules can register a common proxy service as an indirection in between the service exposed to applications and the back-end implementation. As a result, co-modules can seamlessly switch between either of the two implementations (if the hardware resource is available) and provide seamless dynamic acceleration. If the service is implemented as an OSGi *ServiceFactory*, it can even selectively accelerate the service only for certain consumers and serve requests from other bundles through the software implementation. Figure 2 shows a structural overview of both a hardware module (a module containing only a hardware implementation of a service) and a co-module.

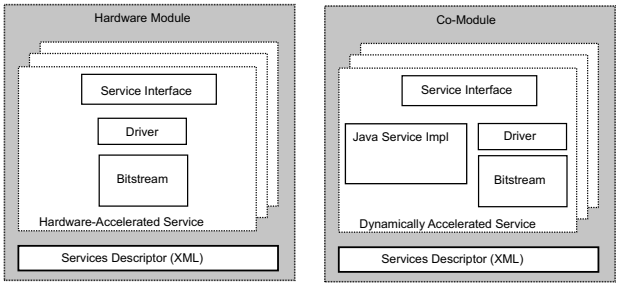

Fig. 2. Structure of hardware-accelerated OSGi services

3.2 The FPGA Bundle Extender

The basic unit of modularity in OSGi is the bundle. Even though there are very few requirements for a bundle to participate in an OSGi application, in practice there is a small piece of code in the bundle which interacts with the runtime and registers or consumes services. This code is specific to OSGi whereas most of the remaining bundle code is standard Java. For some applications, however, this OSGi-specific code is a liability. For instance, considering a web application server based on OSGi, every web application is preferably a bundle and registers its servlets as services. In traditional Java EE, however, web applications are packaged in WAR files, which are JAR files with a set of specific XML configuration files. The requirement to write the boilerplate code so that the servlets are discovered from the `web.xml` file and registered as services so that the server engine becomes aware of their existence is a burden for the adaption of the OSGi model for web applications. The solution to the problem is the extender pattern.

In the extender pattern, there is a singleton entity—called extender—in the system that listens for newly installed bundles. Whenever a new bundle is installed, it scans the content of the bundle for the existence of a specific configuration file. If such file is present, the extender interprets this file and extends the bundle by, for instance, registering services on behalf of the bundle. Thereby, in principle plain WAR files can be used within an OSGi deployment; the extender takes care of integrating the content of the file into the application server.

For Juggle, a similar approach is taken. An FPGA extender listens for new bundles containing a configuration file for hardware-accelerated services and then registers the service on behalf of the bundle. Listing 3 shows an example of a configuration file. Each bundle can contain arbitrarily many hardware-accelerated services. As for traditional OSGi services, properties can be attached to the service on which clients can filter their requests. Instead of selecting either the Java or the FPGA-based service, the FPGA extender generates a service proxy from the service interface. The purpose of the proxy is to provide the system with an interception point located between the caller and the service. This enables the system to seamlessly switch between a software service and a hardware service as well as tracing service invocations to derive performance information.

```
<?xml version="1.0"?>
<co-module xmlns="http://flowsgi.inf.ethz.ch/comodules">
  <accelerated-services>
    <accelerated-service interface="math.AddService">
      <java-service>math.MathAddImpl</java-service>
      <fpga-service>
        <driver>math.MathAddDriver</driver>
        <bitstream>opb_prr_0_adder_partial.bit</bitstream>
      </fpga-service>
      <service-properties>
        <entry key="version" value="1.0.0"/>
        <entry key="foo" value="bar"/>
      </service-properties>
    </accelerated-service>
  </accelerated-services>
</co-module>
```

Listing 3. Juggle service descriptor example

4 Juggling Software and Hardware-Accelerated Services

Not only is the FPGA a singleton entity in the system, the resources of the FPGA in terms of logic gates are also limited and permit—depending on the complexity of the service—just one or a small number of hardware-accelerated services to co-exist at any given time. It is hence inevitable to make resource scheduling decisions and set priorities. For this purpose, Juggle continuously traces service invocations and assembles statistical data to make decisions which services can run in software and which can profit from hardware acceleration.

Deciding which service of a single application to swap into hardware is a policy decision and can thus be best made by the application itself. On an OSGi runtime and particularly on mobile devices, however, it is not unusual to run multiple applications simultaneously. Deciding in favor of a specific application hence requires coordination. However, global knowledge about the setup is against the principle of modularity; a module should only reason locally and not require knowledge about other modules installed on the same system beyond declared or loosely-coupled dependencies.

Juggle deliberately avoids implementing policies and instead expects the platform to implement a *controller* defining the criteria to be used for reconfiguration. For instance, such a policy could be that the application currently running in the foreground and having the focus of the user is prioritized over the background applications. Which service to prioritize could therefore be determined by the window manager, which is by definition an entity with full knowledge of all modules currently using its services. What the system has to provide is access to the basic collected performance data of each service, such as invocation frequency, average duration of the invocations, etc. and a simple imperative command interface to turn a software into a hardware-accelerated service and vice

versa. This command interface consists of a single primitive: the *juggle* operation. As arguments, the juggle operation takes the service id of a service to turn into a hardware-accelerated service as well as the ids of previously hardware-accelerated service to turn back into software services.

4.1 Reprogramming the FPGA

When the controller has issued a juggle operation, the system first does a sanity check, e.g., if the freed slots are adjacent and if the reclaimed FPGA space is sufficient to accommodate the new service. If the check passes, the hardware can be reprogrammed. The Xilinx FPGAs are able to perform a *glitchless* dynamic reconfiguration. This means, if a resource on the chip—despite being in the reconfigured ares—is not affected by the reconfiguration it can be accessed without interruption. There can, however, be problems in the design phase of hardware service implementations. If the place-and-route tool does not have to meet any communication constraints between two hardware services the signal will much likely cross the boundaries of the partial reconfiguration area where the best timing can be achieved. The routing can be different for every hardware service implementation.

The solution is the usage of bus macros [3] which can be seen as fixed data paths for signals going between PRRs. Bus macros serve the purpose of a socket where the corresponding hardware service can be plugged into the system. Hence, they provide the means of locking the routing between hardware services and the static part, making the modules pin compatible with the base design. In addition to locking the routing path, bus macros also serve as switches to enable and disable the transmission of signals. The signal propagation has to be disabled during reconfiguration, and enabled after, to avoid bus congestion or even a corruption of the bus during the reconfiguration process.

In the Juggle design, the bus macros for PRRs are encapsulated into their own IP core (a reusable unit in the hardware design process), the socket bridge. In our prototype system, the socket bridge is controlled over the Device Control Register (DCR) bus. This bus bypasses the standard memory bus and bus controller for low latency and implements a daisy-chain architecture propagating the signals to all attached cores. The communication between the runtime system and this IP core happens through a kernel-driver in the operating system. In our prototype system, we use Linux and have developed a driver which registers a character device to accept control words for opening or closing the socket bridge. When reading from the character device, the current status of the bridge can be retrieved. Since character devices in Linux are represented through ordinary file descriptors, the Java VM can access them as random access files.

The actual reconfiguration happens through the internal configuration access port (*ICAP*) interface. The ICAP device is supported under Linux by a driver in the patched Xilinx kernel and can therefore also be accessed from Java. The system has to open the socket bridge for the RPP, retrieve the partial bitstream from the bundle, write it to the `/dev/icap` character device, and close the socket bridge again. Subsequently, the JNI driver for the hardware service is loaded. If

the service is hardware only, the driver is now registered as an OSGi service under the designated service interfaces. For co-modules, there is already an existing service proxy which needs to be altered to redirect calls to the JNI driver and hence to the hardware service. In order to perform juggling from the software to the hardware service in a consistent manner, all pending service method calls that are still accessing the software implementation will run to completion in software whereas all newly method calls use the hardware. This is consistent with the behavior of dynamic code modifications in the JVM like the *RedefineClasses* function in the Java Virtual Machine Tool Interface (JVMTI) which is frequently used in runtime debugging tools or for runtime aspect-oriented programming (AOP) support.

5 Juggle Prototype System

Our prototype system uses the Xilinx XUPV2P development board [19] containing a Virtex-II Pro FPGA with a total number of 30,816 programmable logic cells. In addition, the FPGA chip contains two embedded PowerPC 450 cores running up to 300 MHz. The cores can have an instruction and data cache with up to 16 KB each and a MMU. Xilinx provides a patched Linux kernel tree that runs on the PowerPC cores.

The system boots off a flash device containing the system ACE file which initially configures the PPC cores as well as programming the static parts of the FPGA required to connect the PPC cores to the peripheral hardware. The PowerPC405 program counter is set to the starting address of the Linux kernel also contained in the ACE file. During and after the boot process the kernel can use the flash card as a secondary storage device for its root file system.

Figure 4 shows a block diagram of the base system design used for the prototype system. A single PowerPC core is attached to a Processor Local Bus (PLB) which is part of the IBM CoreConnect Bus Architecture specification [4] and in this system serves as a communication backbone. An Ethernet connector, a serial port, and the compact flash connector for the card holding the system ACE file are attached to this bus through their controller logic cores. In addition, there is a bridge which connects to a second bus, the On-Chip Peripheral Bus (OPB). Even though this bus type is deprecated in recent versions of the Xilinx tools, the Virtex-II Pro internal configuration access port (ICAP) is only capable of communicating through the OPB. Later versions of Virtex chips feature ICAP devices that can be directly attached to the PLB. Our prototype contains only a single PRR, for the proof of concept, which is attached to the OPB through a socket bridge. This has the consequence that only a single hardware-accelerated service can run at any time.

The figure shows the logical structure of the base system. Physically, the PR region of the system has been placed at the right edge of the chip. The reason is that the entire memory bank is connected to the left side of the chip so that choosing the right side for the PRR keeps the the number of static routes crossing the module boundaries low. As a consequence, the RP region can cover

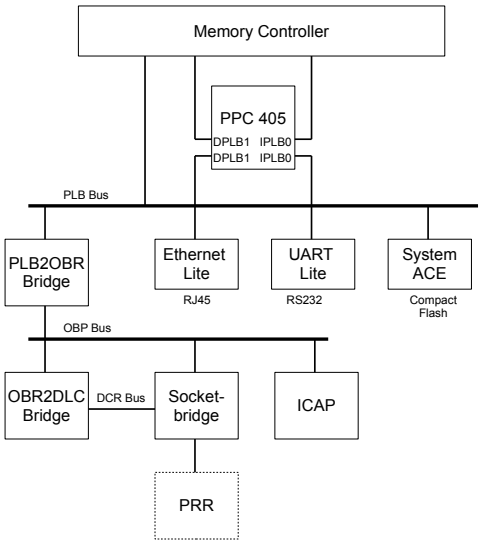

Fig. 4. Base design of the prototype system

almost the full height of the device except for four rows of IOB and IOI at the top and bottom. The width of the PR region spans 8 CLBs, leading to a total size of almost 16% of the FPGA fabric (Table 1):

	Slice	Mult	Ram16	TBUF
Entire FPGA	13696	136	136	6848
PRR	2240 (16.35%)	20 (14.7%)	20 (14.70%)	6848 (16.35%)

Table 1. Physical resources of the FPGA chip and the PRR.

The prototype runs the PowerPC core at 300 MHz and features 256 MB of external DDR SDRAM. As an operating system, it uses the patched Xilinx Linux kernel based on version 2.6.35 and a Java virtual machine (three different VMs have been successfully tested). After the system has booted, about 190 MB remain available for user-space programs such as the JVM and Juggle. Due to the constrained resources, Juggle relies on an updated version of the highly optimized Concierge [13] OSGi technology implementation. The OSGi framework is enhanced with support for hardware-accelerated OSGi bundles through an FPGA extender. The prototype system does not feature an autonomic controller for juggling software and hardware implementations of services. Instead, it registers an extension service for the Concierge shell so that the juggling can be triggered on demand by the user of the system.

6 Evaluation

The use case for evaluating Juggle is an application that requires encryption. This can, e.g., be the encryption of data on the internal storage of the device or a secure protocol which encrypts the data before transmission. Normally the encryption functionality would either be implemented in software or, if performance critical, in hardware as an ASIC. Security, however, is one of the areas that require a constant update of the technology used due to bugs in implementations and exploits through weaknesses in the algorithms. For instance, if the mobile device was shipped with an ASIC accelerating the encryption of data with the Data Encryption Standard [10] (DES), the de-facto standard until 2004, it would be obsolete by now since with todays possibilities the DES encryption cannot be considered secure. If, however, the encryption is implemented as a hardware-accelerated service, the device becomes much more flexible and future-proof. First, the encryption algorithm can be exchanged at any time, e.g., with a Triple-DES encryption [11], even at runtime. Second, encryption can selectively run hardware-accelerated, e.g., when the performance of the interactive process is limited by the encryption of data. An example would be a user decrypting an larger email message. When the user switches the foreground task, e.g., to the music player, the audio decoding becomes the hardware-accelerated service and any encryption happening in the background runs through the software implementation of Triple-DES.

6.1 DES and Triple-DES as Hardware-Accelerated Services

A DES and a Triple-DES encryption service have been implemented as hardware-accelerated OSGi services for Juggle. Both services are implemented in software—using either the Java Cryptography Extension (JCE) provider shipped with the VM or the BouncyCastle [16] Java library—and in hardware in the form of a partial bitstream for the PR region in the base system. Listing 5 shows the common service interface of both DES and Triple-DES so that one can be easily exchanged with the other.

```
public interface EncryptionService {
      void loadKey (byte [] key);
      int encrypt(ByteBuffer data, int size);
      int decrypt(ByteBuffer data, int size);
}
```

Listing 5. Interface of the EncryptionService

The bitstreams for hardware-accelerated services for a given PRR have the identical size (≈145 kB in case of our prototype system) since the entire PRR is reconfigured in either case. In practice, tools for partial reconfiguration like

Xilinx PlanAhead create compressed bitstreams so that there can be slight variations in the size depending on the complexity of a design. Table 2 shows a detailed resource consumption of the two hardware implementations when programmed into the FPGA. As reference points we have added two simple hardware-accelerated services we used during the development of the system, an adder service and a multiplication service, which both take two Java primitive type integers as input and return the result of the arithmetic operation as a Java integer. The Triple-DES implementation in fact contains three instances of DES and hence consumes about three times more chip real-estate than the DES implementation. Both hardware-accelerated services leave enough space so that potentially other services could run in parallel if the base system was designed to support this.

	PRR	add	mul	DES	Triple-DES
LUT	4480	90 (2.01%)	58 (1.29%)	1081 (24.13%)	3008 (67.14%)
Flipflop	4480	176 (3.93%)	144 (3.21%)	513 (11.45%)	1527 (34.08%)
Slice	2240	108 (4.82%)	88 (3.93%)	660 (29.46%)	1835 (81.92%)
Mult	20	0	1 (5.00%)	0	0

Table 2. Physical resources used by different hardware-accelerated services

Since the reconfigurable area is always entirely overwritten, the reconfiguration time is solely a function of the target service and not of the service previously located in the PRR. Hence, the time depends on the number of elements to be reconfigured. Table 3 shows the exact sizes of the example bitstreams and the reconfiguration times. The static full bitstream used to boot the system is given as a reference point in terms of bitstream size but it indeed cannot be used for reconfiguration. The reconfiguration time for our examples varies between 11.2 and 24.9 milliseconds. Values reported in the literature indicate that in general the reconfiguration time of the Virtex-II Pro is between 10 and 35 milliseconds. There is an additional overhead involved in switching the socket bridge, which adds on average about three milliseconds. In total, the time to juggle a service is hence between 15 and 30 milliseconds for our examples. This indicates that an on-demand reconfiguration is feasible given the latency requirements of applications typically found on mobile devices.

The time to create the initial hardware service is in range of 100 milliseconds for our DES and Triple-DES example and includes the time to create the service proxy and the time to load the JNI driver for the hardware. Loading either of the two encryption bundles, which are implemented as co-bundles and also register a software service, takes less than a second. The startup time of Juggle, which includes the startup time of the Java virtual machine, Concierge with the basic bundles, and the load time of the FPGA extender bundle, is on average 6 seconds in the prototype setup.

bitstream	size in bytes	reconfiguration time
static full.bit	*1448817 (100%)*	—
add	123249 (8.50%)	11.159 msec
mul	128222 (8.85%)	24.896 msec
des	149087 (10.29%)	13.441 msec
tdes	149088 (10.29%)	13.374 msec

Table 3. Size of the bitstream and reconfiguration time

6.2 Acceleration through Juggle

Dynamic juggling of software and hardware services has shown to be feasible for a large class of applications. What remains to be shown is that hardware services have in fact a significant potential for speeding-up Java programs. For this purpose, we have evaluated different ways of performing TripleDES encryption on our prototype board using different Java virtual machines.

For the PowerPC architecture, there is no implementation of the original Sun Java HotSpot Virtual Machine. However, since the sources were released to the open source community, the IcedTea project [5] has implemented a portable version of the OpenJDK which is largely free of assembly code (IcedTea Zero) and has been successfully ported to the PowerPC and other architectures. The main caveat of the IcedTea Zero VM is that it is purely interpreting and does not feature just-in-time compilation. Hence, the performance of applications running on the IcedTea Zero VM is significantly lower than on a JIT-enabled virtual machine. For comparison, we have taken a version of the IcedTea VM enhanced with the Cacao [8] just-in-time compiler. Both versions are based on the same Java 6 version 1.8.2 build 18 of IcedTea. The Zero VM is version 14.0-b16, the Cacao JIT corresponds to the released version 0.99.4. The third virtual machine used is the IBM J9 VM for PowerPC. This virtual machine has JIT and is typically used in production servers of the PSeries but also runs on the PPC 405. The version used is J2RE 1.6.0 IBM J9 2.4 build pxp3260-20071123_01.

As a first reference point, we have measured the performance of different Java TripleDES implementations on the three virtual machines running on our evaluation prototype system. In general, cryptography for Java applications is supported through the Java Cryptography Extensions, an API for data encryption, authentication, and key management. All three virtual machines used in the experiments ship with a JCE provider. The providers of the two IcedTea flavors are identical while the IBM implementation is based upon a different code-base. For further comparison, we have used the open-source JCE provider *BouncyCastle* [16] in its latest version 1.38 for Java 6. This library can uniformly run on all three virtual machines.

The experimental setup for this and all following experiments is the encryption of a buffer filled with random bytes, using TripleDES with the same fixed key. The size of the buffer is varied in the experiments to get an impression of the overall performance characteristics of the corresponding implementations. Fig-

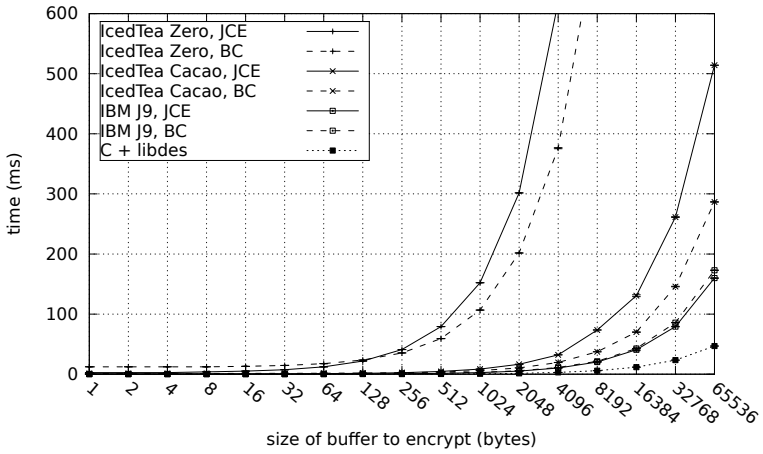

Fig. 6. Performance of Triple-DES encryption in Java

ure 6 summarizes the experimental results for the various Java implementations. As a baseline of comparison, the graph additionally shows the performance of a C program using the Triple-DES implementation from Eric Young's *libdes*.

The first conclusion to be drawn from the results is that the interpreting VM has a significantly lower performance than the other two VMs. This can be expected since cryptography is computation-intensive and can hence profit to a large extent from just-in-time compilation. The IBM VM performs better with its own JCE provider whereas on Cacao BouncyCastle performs better than the built-in OpenJDK provider. Overall, however, even this implementation is still almost a factor of three slower than the C implementation, which is surprising given its JIT but it is possibly constrained by the available systems resources.

The next experiment compares the performance of the two software implementations (IBM J9 JCE and C+libdes) with the performance of a hardware-accelerated Juggle service running on the different VMs. The hardware-accelerated service consists of a Java interface, a JNI driver, and the corresponding Triple-DES logic in the FPGA.

The JNI driver exchanges data with Java through ByteBuffers and shuffles data to the FPGA TDES core by writing to software registers. Triple-DES is a block cypher. For the encryption of each block (of 8 bytes), the JNI driver writes the data into two 32 bit registers of the TDES core and then alters a status register to indicate that the data is ready and the requested operation is an encryption. When the core has encrypted the data, it sets the status register to a success value. The JNI driver busy-waits on the content of the status register and then reads back the encrypted result from two 32 bit registers. This design has been mainly chosen for simplicity, more sophisticated implementations might further improve the performance of the hardware. The TDES core runs with the

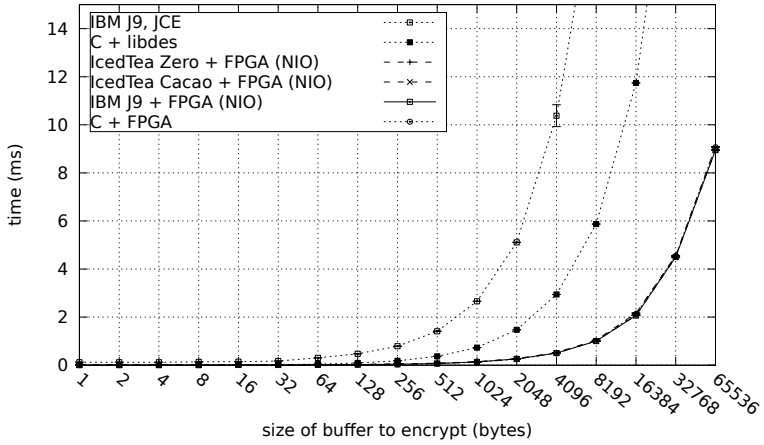

Fig. 7. Performance of Triple-DES encryption through a hardware-accelerated service

bus clock speed, which is 100 MHz in the prototype system and therefore a factor of three lower than the CPU clock.

Figure 7 shows the measured results and, as a baseline, the performance of a C implementation using the same FPGA TDES core for acceleration. Hardware-accelerated encryption in Java can provide a performance equal to using the FPGA directly from C since the static overhead becomes irrelevant for realistic buffer sizes. When encrypting buffer of at least 64 bytes size, a Juggle hardware service accelerates the encryption by a factor of almost 20 compared to the best Java software service. Furthermore, when accelerating the performance-critical code through hardware implementations, the interpreting VM can reach almost the performance of a JIT-enabled VM. This is an interesting design option for highly resource-constrained systems that cannot afford the memory and storage footprint of a just-in-time compiling VM.

6.3 Power Consumption

Besides performance, power consumption is a major issue for mobile and battery-powered embedded devices. Therefore, we evaluated the power consumption of three different implementations of TDES by using a wattmeter introduced between the power supply of the prototype board and the power outlet. Hence, the values measured determine the consumption in the primary circuit and correspond to the de-facto consumption observed by an operator of the device. For benchmarking, we used the TDES encryption of a buffer of 1024 bytes in 30 runs of loops of 1000 encryptions and measured the power consumption for the OpenJDK IcedTea VM with the Cacao JIT and the JCE encryption provider, for the C implementation using the libdes library, and for the same OpenJDK IcedTea/Cacao VM but using the FPGA for the encryption. Figure 8 shows

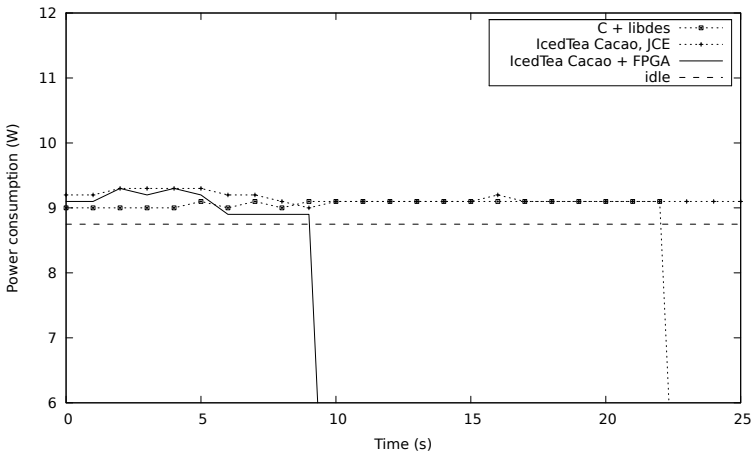

Fig. 8. Power consumption of the different TDES implementations

the results for the three different implementations. The power consumption of the board in the idle state is 8.75 W, illustrated by the dashed horizontal line. The two Java implementations have a peak consumption in the first six seconds, which is the time that the JVM takes to start. After this startup time, the software implementation has a relatively stable power consumption of 9.1 W whereas the FPGA-based implementation uses only 8.9 W. The C implementation has a constant consumption of 9.1 W.

Integrating the power consumption over the runtime of the test run gives the total energy required for performing the encryption task. The pure Java implementation, which has a total runtime of 346 seconds (the figure does not show the entire runtime for Java/JCE), consumes 3151 Joule. With the C implementation, the device consumes a total of 199.6 Joule, only slightly more than 6% of the energy for Java/JCE, mainly due to the significantly lower runtime. When accelerated through Juggle, however, the encryption can be performed in Java using only 81.7 Joule, which is about 41% of the energy spent with the C implementation.

7 Related Work

Juggle is not the first attempt to interface between a high level language like Java and reconfigurable hardware. JBits [1] is a set of Java libraries that can read bitstreams either generated by the toolchains or from a currently running FPGA device. It provides an API to modify a configuration bitstream and use it to reprogram the device. Unfortunately, JBits provides little abstraction over a hardware description language. Hence, it is highly platform-dependent and requires the using application to explicitly deal with low-level details such as the routing.

Liquid Metal [2] features the Lime language which is based on Java but extended with a special and more restricted type system amenable to bit-level analysis. Lime can be compiled both into Verilog and successively into FPGA bitstreams as well as to Java byte-code. The target domain of Liquid Metal is similar to Juggle as both systems target devices with both a conventional CPU and an FPGA as an additional resource for dynamic acceleration. The major difference is the level on which the systems operate. Liquid Metal attempts to create a unified language for both the software and the hardware design. Juggle in turn focuses on the integration of the two worlds through composition of modules.

A different approach has been taken with JOP [14], a Java optimized processor implemented on top of an FPGA. The motivation of JOP is to enable the use of high-level productivity languages like Java for programming FPGA chips. The authors point out that the programming languages usually used on top of systems on a chip like C and Assembler provide poor abstractions to the programmer. The result of this consideration is a JVM implemented as a processor on an FPGA. The original Java byte-code is translated by the processor into an address in the own microcode format of the FPGA-driven JVM.

Ullmann et al. [17] present a complete approach to a module based architecture for automotive control devices. Todays automobile classes contains up to 100 control devices which quickly obsolete and decreases the product life cycle from 5 to 2 years. The adaptivity of reconfigurable devices can increase the product life cycle while reducing the cost and risk for development and later maintenance.

A high-level approach for using the Xilinx FPGA reconfiguration is explained in the work of Williams et.al. [18]. They present a modular platform for RSoC called Egret designed around the idea that complex systems can and should be designed by composition. The specification of an assembled hardware module stack is given to a software tool that constructs the appropriate FPGA configuration, as well as software infrastructure such as device drivers.

8 Summary and Discussion

Juggle shows that a modular and loosely-coupled approach to integrating software and reprogrammable hardware facilitates a flexible and dynamic co-existence between software and hardware services. Applying the same management facilities to both worlds simplifies the development of such systems and at the same time gives the maintainer the opportunity to alter the setup even at runtime. OSGi is extensible enough to be used for this purpose and the extender pattern avoids large parts of the boilerplate code required for registering services. The latency of reprogramming reconfigurable areas is low enough to seamlessly switch between the software and hardware and accelerate the most critical tasks at any time. Due to the common interface that Juggle applies to both software and hardware services, the substitutability principle of modularity ensures that existing applications do not need to be modified for Juggle. Hence, their per-

formance can gradually be improved through introducing hardware services. In practice, the degree of acceleration can be significant, as shown with TripleDES where we reached a speedup of 20 despite the unoptimized hardware design.

References

1. Guccione, S., Levi, D., Sundararajan, P.: JBits: Java based interface for reconfigurable computing. In: MAPLD '99 (1999)
2. Huang, S.S., Hormati, A., Bacon, D.F., Rabbah, R.: Liquid metal: Object-oriented programming across the hardware/software boundary. In: ECOOP '08. pp. 76–103. Springer-Verlag, Berlin, Heidelberg (2008)
3. Huebner, M., Becker, T., Becker, J.: Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In: SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design. pp. 28–32 (2004)
4. IBM Microelectronics: CoreConnect Bus Architecture. `https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture` (1999)
5. IcedTea Project: OpenJDK IcedTea. `http://icedtea.classpath.org` (2006)
6. IEEE: Standard VHDL Language Reference Manual. ANSI/IEEE Std 1076-1993 (1994)
7. IEEE: Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language. IEEE STD 1800-2009 (2009)
8. Krall, A., Grafl, R.: CACAO - A 64-bit JavaVM Just-in-time Compiler. Concurrency: Practice and Experience 9, 1017–1030 (1997)
9. Mérillon, F., Réveillère, L., Consel, C., Marlet, R., Muller, G.: Devil: an IDL for hardware programming. In: OSDI '00 (2000)
10. National Institute of Standards and Technology: Data Encryption Standard (DES). FIPS Publication 46-2 (1993)
11. National Institute of Standards and Technology: Data Encryption Standard (DES). FIPS Publication 46-3 (1999)
12. OSGi Alliance: OSGi Service Platform, Core Specification Release 4, Version 4.2, Draft (2009)
13. Rellermeyer, J.S., Alonso, G.: Concierge: A Service Platform for Resource-Constrained Devices. In: EuroSys '07: Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems. pp. 245–258. ACM (2007)
14. Schoeberl, M.: JOP: A Java Optimized Processor. In: On The Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real- Time and Embedded Systems (JTRES 2003). vol. LNCS 2889, pp. 346–350 (2003)
15. Sony Corporation: Virtual Mobile Engine (VME). `http://www.sony.net/Products/SC-HP/cx_news/vol42/pdf/sideview42.pdf` (2002)
16. The Legion of the Bouncy Castle: Bouncy Castle Java Cryptography APIs. `http://www.bouncycastle.org/java.html` (2000)
17. Ullmann, M., Huebner, M., Grimm, B., Becker, J.: An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. International Parallel and Distributed Processing Symposium 4, 135a (2004)
18. Williams, J.W., Bergmann, N.W.: Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip. In: ERSA. pp. 163–169 (2004)
19. Xilinx Inc.: Xilinx University Program, Virtex-II Pro Development System, Hardware Reference Manual. `http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf` (2009)