University of Southampton

Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

Core Services: A new design methodology for MPSoCs

by

Dimitrios Kouzis - Loukas

September 2006

A dissertation submitted in partial fulfilment of the degree of
MSc Microelectronics System Design
by examination and dissertation

Abstract

Significant research effort in platform based design has given numerous interesting and innovative solutions to some of the recent VLSI design automation problems. Emerging Multi-Processor System-on-Chips (MPSoC) feature reconfigurable components and hierarchical busses or Networks-on-Chips as communication infrastructure.

Core Services methodology reported in this dissertation uses mechanics inspired by Web Services that most software engineers are already familiar with to exploit efficiently dynamic partial reconfiguration and run-time mapping of current System-on-Chips (SoCs) to provide guaranteed performance increase and fault tolerance on-demand. Core services can be efficiently implemented in platforms with communication infrastructures including busses and network-on-chips.

Core Services define a function-level abstraction of the underlying hardware processing elements and a resource management mechanism (Service Broker) which optimises at run-time the mapping of functionality to available processing resources. Service Broker also measures the frequency of requests and configures reconfigurable elements to increase system's performance. Fault tolerance is considered as a resource management problem and thus solved transparently by the Core Services framework.

We validate Core Services methodology by applying it on Xilinx's reconfigurable platform for high-end FPGAs. The stack of software and hardware components for communication, data management and function virtualization is implemented and evaluated. A user-friendly application interface (API) and a powerful device driver for MontaVista embedded Linux are provided. Hardware and software components are created automatically by an easy to use platform building application able to run on Windows and UNIX workstations. The platform is being evaluated with two computationally intensive applications, AES encryption and MP3 decoding that get accelerated in different levels of granularity. We conclude by presenting our benchmarking results on a complex use case of these applications.

Acknowledgements

First and foremost I would like to my supervisor Professor Bashir Al-Hashimi for his invaluable support and patient guidance during the course of this thesis.

I would also like to thank Dr. Paul Rosinger for his support on my first steps of this thesis. Our conversations gave me insights on unexplored fields and revealed interesting problems. It was a pleasure to work with you.

I dearly thank my family and my dear Eva for their never-ending love and moral support.

Table of contents

List of figures

Chapter 1. The landscape

We live in the System-on-Chip (SoC) era. Complex requirements of current applications have forced VLSI design engineers to integrate multiple components on a single chip. Efficient communication between the components is challenging and has been the subject of intensive research. The Network-on-Chip (NoC) communication scheme seems promising in satisfying the communication needs of current SoCs. The structure of a modern SoC featuring NoC can be seen in Figure 1.

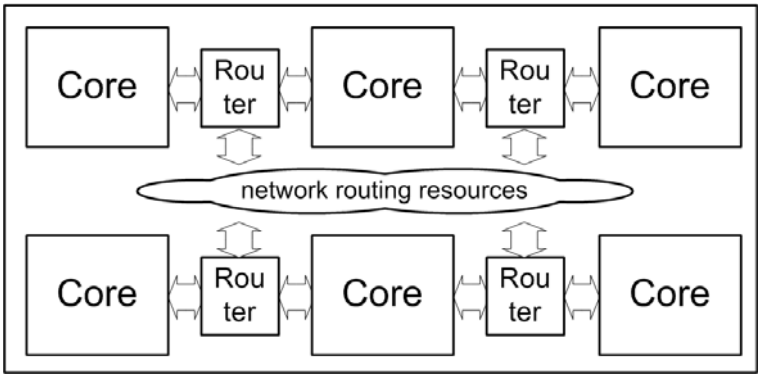


Figure 1. A System-on-Chip (Soc) featuring Network-on-Chip (NoC)

Figure 2 shows the percentage of publications that include terms used in modern VLSI design such as reconfigurable, NoC, SoC and also Fault Tolerance. This figure's data was obtained using the IEEE Xplore over the period 1990-2005.

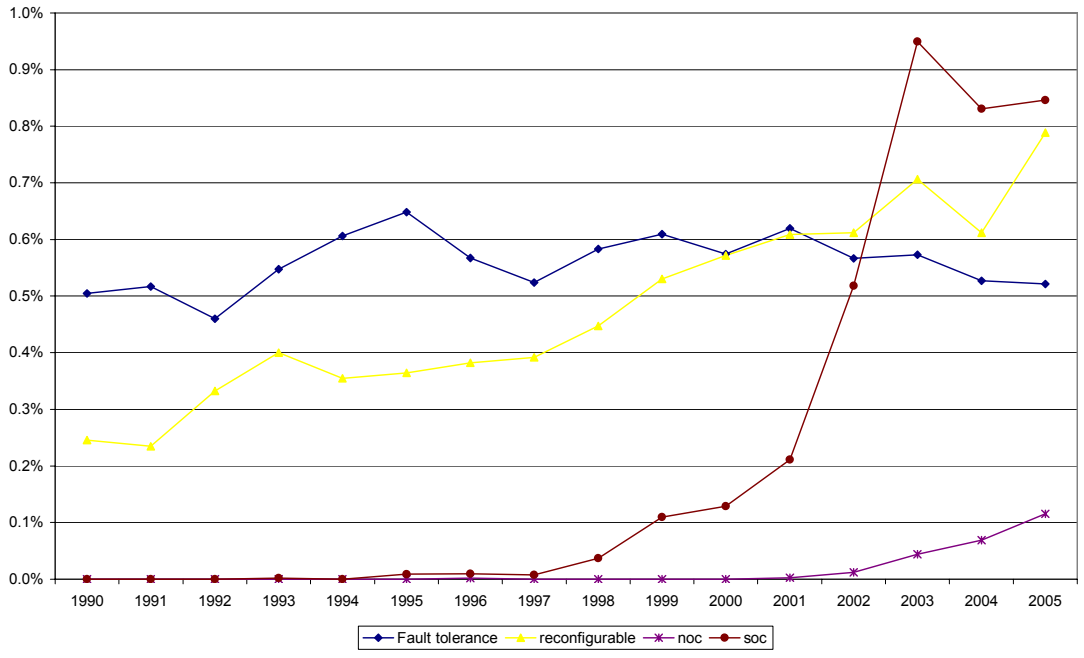


Figure 2. Normalized publications that include certain terms on IEEE Xplore

We can see that after 2003 about 1% of the annual publications include the term "system on chip". Impressive considering the different research areas IEEE Xplore covers. The silicon integration and new opportunities that current system-on-chips

provide is the hottest topic of the first half of this decade and will probably continue to be. We can also see that there is a steady increase of publications on reconfigurability. As clock frequencies are hard to increase anymore researches are looking at reconfigurability as a means of increasing performance by transforming the performance problem from clock frequency problem to an area problem were Moore's law (see section 1.1) still holds true. We can also observe the emerging field of network-on-chips being in 2005 in about the same position where system-on-chips were in 2000. A final observation can be made for fault tolerance. For the last 15 years 0.5% of all the publications on IEEE Xplore refer to fault tolerance. The reason is the wide meaning of this word, making it useful on many contexts but clearly there is a continuous need for fault tolerant systems in all levels of Electrical and Electronics Engineering.

The key message from Figure 2 is clear. We are now designing systems-on-chip that have multiple processing elements possibly reconfigurable and in the very near future innovative designs will deploy interconnection infrastructures such as networks-on-chip. On the following sections we will present the current status of SoC design regarding the processing elements (section 1.1), communication infrastructure (section 1.2) and reconfigurability (section 1.3). We will conclude in section 1.4 with some future perspectives.

1.1 Multiprocessor System-on-chips

There is a constant need for integrated circuits with more processing performance and lower power consumption. At the same time it is increasingly hard to increase the operation frequency or lower the supply voltage without affecting system's reliability. The answer to these constraints seems to lie in Moore's "law" [1]; "the number of transistors on a chip duplicates every 24 months". The International Technology Roadmap for Semiconductors predicted that chips with billion transistors where within reach [2] and Montecito version of the Itanium processor already proved that in 2006 by having 1.7 billion transistors.

More transistors on a chip allow us to build complex systems optimized for certain application domains. More processing performance and lower power consumption is realized by using specialized processing elements (PEs) for tasks with different requirements. For example on a single chip a DSP algorithm can run on a DSP core, an operating system on a microprocessor and advanced video operations on custom hardware. Increased complexity makes designers face many challenging problems [3]. Power consumption increases with the number of components and may be attacked at device level [4], communication level [5] and software level [6]. The real-time constrains and performance (throughput) constrains are attacked by employing innovative heterogeneous architectures. For example instead of the standard memory hierarchy (registers, caches, external memory) employed on general purpose computational systems a MPSoCs frequently features custom memory setups including FIFOs, caches and scratchpad memories. These advanced architectures give competitive advantage on MPSoCs over traditional symmetrical multi-processor systems in terms of performance and power consumption. Security is an emerging issue for MPSoCs. Hardware and software must be designed to be secure especially in mission critical applications. This overlooked design aspect has to be given special attention since more SoC based devices are connected to public networks like the Internet. Finally, the most important problem with the increased system complexity is the ever increasing design gap. Every MPSoC requires its own suite of software tools (compilers, simulators etc.) and testing and verification becomes increasingly complex, time consuming and expensive. The solution seems

to be in design reuse with a platform based design approach which is a good choice but not a panacea [7].

1.2 On-chip communication

Many cores mean more communication between them and between the shared resources. This may be significant when blocks of data have to be transferred especially if they are small and frequent in which case hidden factors like handshaking and arbitization may produce significant overhead which is not visible unless detailed models of communication get used.

Many solutions have been reported to the problem of communication. Traditionally busses are employed for this purpose. An ASIC designer had to design carefully a custom set of busses to meet the bandwidth requirements of each communication path. These busses where very often bi-directional, using tri-state buffers. With higher frequencies bus design and verification became increasingly difficult because wires get longer and the inductive characteristics make transmission line effects apparent. The amount of design re-use is also limited since protocol translators (wrappers) have to be used whenever there is a change on the bus topology which costs in terms of development time and system's performance.

It became apparent that some standardization of the bus interfaces would benefit the ASIC industry. Three are the most standard bus interfaces at this moment. IBM's CoreConnect™ bus, ARM's AMBA bus and OpenCore's Wishbone interface. The first two come with some out-of-the-self implementations of interconnection and arbitization scheme for certain platforms while the third one leaves interconnection to the designer. CoreConnect and AMBA are both hierarchical busses featuring high performance busses for components such as memories and processors and lower performance busses for slower peripherals like communication ports. High speed buses tend to be unidirectional because bidirectional registers are hard to implement and registers are required because data transfers are pipelined in more that one bus clock cycles. High performance busses use multiplexers and have separate input and output paths that unfortunately use more wiring and area. Busses don't scale up so well because only one master can own the bus at each moment.

The successors of busses seem to be networks-on-chip. The idea of NoCs appeared at mid 2001 by the classic works of Benini and De Micheli [8, 9]. A thorough overview of the NoC technology its promises and details for two NoC implementations Xpipes and Æthereal can be found in [10] and other recent are listed in [11]. The introduction of NoCs forced designers to move from a computation-centric to a communication-centric approach. New models [12, 13] and tools [14] had to be developed to aid design exploration [15] and hw/sw co-design [16].

Network-on-Chips allow simultaneous use of chip's resources by having cores or local busses assigned to a node on an on-chip network. Nodes can communicate to each other by transferring packets of data. A response to a request from a core may take several clock cycles to arrive but it may be large enough to pay for this latency. At the same time many other cores may be taking equally large responses from other sources if there are no routing conflicts. Networks-on-chip have the potential of making better use of the increasingly expensive global wiring of a chip by utilizing less expensive routing logic. Interestingly only wishbone interface [17] mentions crossbar switch interconnections (the nucleus of every network-on-chip topology). All busses can support network-on-chip topologies via special network interfaces as seen in [18]. By issuing non-blocking memory writes on NI a master core (MC) makes requests and sends parameter data. Then MC proceeds in doing other tasks

and when the reply to his request is available on the NI it issues an interrupt to the MC in order to notify that reply data are ready. The terminal core completely abstracts the network infrastructure and gives the programmer a relatively familiar programming model if you set aside the fact that the response is asynchronous. Alternatively polling or thread suspend on a multithreading environment can give a synchronous feeling of this communication.

As NoCs is a new technology there are many unexplored fields. For example in [19] a study on the differences between battery efficiency and energy efficiency is being done for reconfigurable hardware. Battery behaviour is non-linear and the energy delivered is a function of the discharge profile. A similar study doesn't yet exist on reconfigurable NoCs. Security in NoCs is also an almost untouched field. In [20] some possible attack scenarios are being examined.

1.3 Reconfigurable hardware

Reconfigurable computing has received interest for more than two decades. Despite of that interest there are very few industrial applications of reconfigurable computing. As with artificial intelligence, the reason that reconfigurable computing doesn't yet seem to have produced impressive results is that it gives techniques to other research areas without being credited.

"Reconfigurable hardware is able to merge the performance of ASICs and the flexibility offered by the microprocessors" [21]. Critical software loops (kernels) can get accelerated [22, 23] by reconfiguration and at different levels of coupling between the processor and the reconfigurable fabric [24] giving different performance/energy efficiency levels in the cost of design complexity. There are various ways to categorise reconfigurable computing (for a more thorough discussion see [25]). The most widely adopted is the fine versus coarse-grained categorisation. The former is referred to our well known FPGAs which have very small reconfigurable datapaths usually 1-bit wide and a large communication mesh that can route them very flexibly. There also exist coarse-grained reconfigurable processors in the form of reconfigurable arrays (RAs) that feature reconfigurable datapaths with sizes larger that 1. These feature less configuration memory, reduced reconfiguration time and complexity of the placement and routing problem but are less flexible than fine grained solutions. In [26] and [27] numerous examples of this category are presented with details about their programming and a discussion of the software/configware partitioning problem. Nowadays we see the two categories merging as for example fine-grained FPGAs feature coarse-grained components like multipliers, DSP primitives, block rams and microprocessors but even more as their building logic blocks are becoming complex enough to model 1-4 bit ALUs.

The most important aspect of run-time reconfigurable processors that is often overlooked in the literature is that they impose, at least up to now, a large reconfiguration time during which they can't perform any computation. This creates a trade-off between how often reconfigurations are decided and the speedup that the reconfigurable hardware provides. Partial reconfiguration [28] reduces reconfiguration time but increases complexity by giving the designer flexibility on the amount of the fabric he wishes to reconfigure. In [29] for example a two level reconfiguration scheme is presented to minimize reconfiguration time. In a mutli-trheaded [30] and even more in a multi-processor environment the use of reconfigurable PEs has to be scheduled accurately in order to maximize hardware's acceleration. Our work goes one step further by considering also communication costs into run-time reconfiguration.

Reconfigurable hardware and NoCs fit suitably well each other. The varying communication channel capacity demand of reconfigurable hardware can be addressed by dynamically reconfigurable NoC as shown in [31-33]. A comparison between this technique and dynamically reconfigurable busses is presented in [34]. Similar approaches have been examined in [35]. In [36] it is observed that NoCs would be an excellent routing resource for FPGAs and such a combined flow is presented in [37].

1.4 The future

The future may not hard to predict. Two different worlds; traditional computing and embedded computing are converging. The problems that computer scientists were facing a few years ago are soon going to be faced by electronic engineers. For example with multiprocessor SoCs well known problems of multithreading and computer clusters like priority inversion, convoying, deadlocks, livelocks and composability [38] are going to be faced on embedded software/hardware. Distributed databases-on-chip (DoC) over networks-on-chip may likely replace the traditional shared memory in order to attack such weaknesses and provide software developers with a familiar programming model hiding the weirdness of the underlying hardware. We already see some recent work [39-42] on the old concept of transactional memory [43, 44] and some new ideas [45].

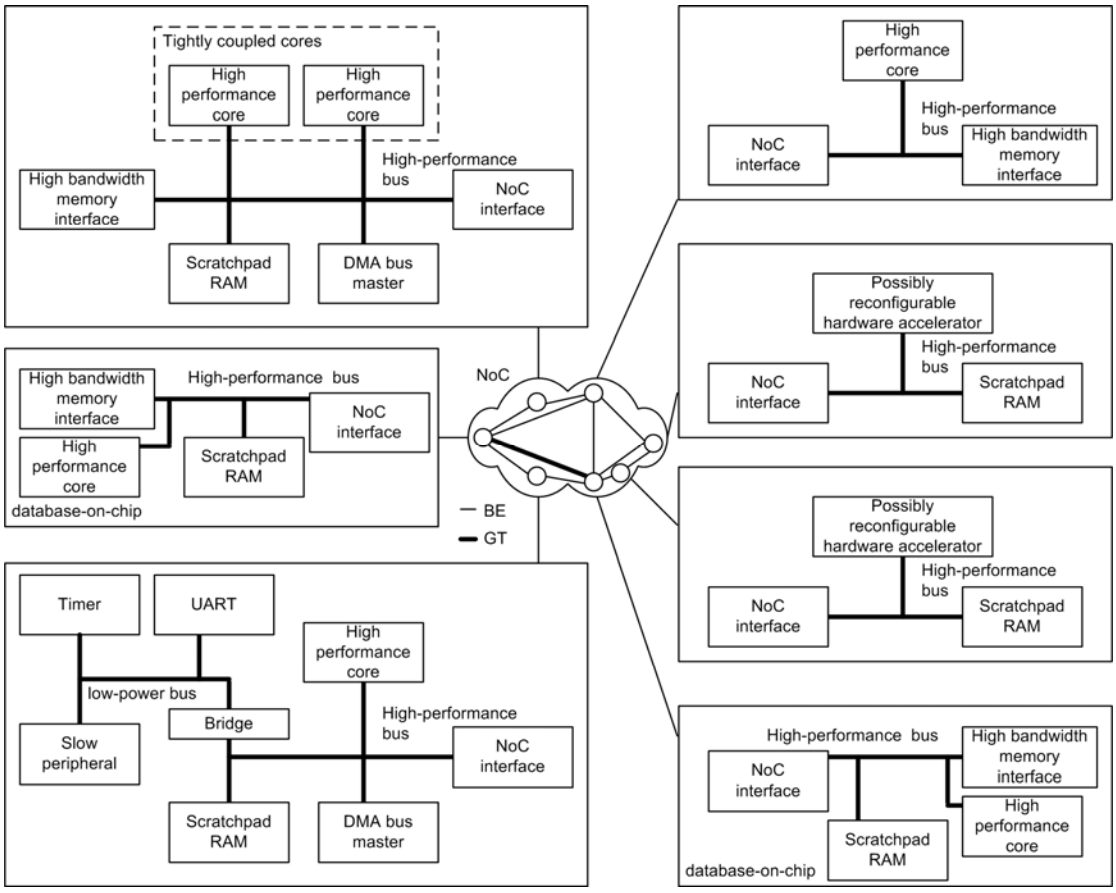


Figure 3. NoC based System-on-Chip

Although there are numerous thoughts at present now on the architectures that will be built around NoCs, the communication patterns that arise in most applications dictate topologies like the one shown in Figure 3. As we can see busses not only don't get abandoned but are used even more but they get simple (again). Increased

round-trip time of NoCs prevents them from fetching instructions on a processor and thus each core must have its own local memory at least for storing the program. The way to attack the memory bottleneck is by using multiple memory interfaces wherever needed. Tightly coupled processors are going to share the same bus while clusters of processors are going to communicate via NoC. The main communication mechanism over NoC is likely going to be Best Effort (BE) for small data packets (<8kB) and thus the buffers on routers are going to consume reasonable area. Strategically based distributed databases-on-chip are going to satisfy the shared memory requirements of cores with a safe way providing transactions and coherence protocols. The inter-processor communication using NoCs is going to be limited to message passing possibly with references on database keys. All the processors will be multi-threaded in order to take advantage of the idle time between network requests and responses.

Chapter 2. Introduction to Core Services

In the following sections we are going to present briefly the Core Services Mechanics (section 2.2) after a short introduction to the inspiring software technology of Web Services (section 2.1). In section 2.3 we are going to present the advantages of Core Services over traditional methodologies and in section 2.4 we present related work highlighting the differences with Core Services.

2.1 Web services

According to the W3C [46] a Web service is a "software system designed to support interoperable machine-to-machine interaction over a network". It power lies in the flexibility it provides. A Java program can invoke Visual Basic .Net functions on the same or another PC that could run the same or another operating system. One may reasonably wonder, is this enough to make a technology such a success? The real reason that web services are a success is that companies found it as an easy way to charge for their services on a per usage basis and at the same time programmers found it reasonably easy to use. For example by using web services, Google Maps can charge a very small fee per request. If web services weren't available it should licence access to the whole GIS database to a client obviously in a much higher rate.

Clearly it would be of benefits in terms of flexibility to deal with expanding number of cores with a similar approach. For example a general purpose microprocessor running Linux could be able to invoke a function on a DSP core or an non-programmable hardware component. Actually it would be even better if we didn't have to statically refer to a component in order to invoke the function. What we want is to have the work done with the best (e.g. fastest) way and we don't really care on who exactly is the core that is going to execute it.

So, lets see the actual mechanics behind the web services. As you can see in Figure 4 there are three main actors for web services; the Service Provider, the Service Requester and the Service Broker. The names are self explanatory except of the Service Broker. This is a repository that holds information about web services from many providers. A service requester can query a service broker and get a list of web services that suit its needs, if available. This is the least developed part of the standard.

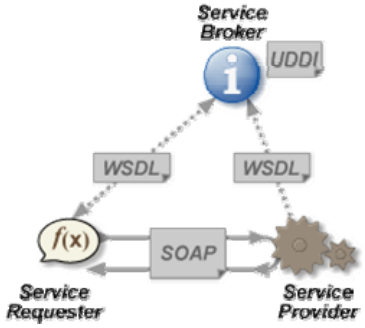


Figure 4. Web service's mechanics

These actors use some protocols to achieve their goals. The service requester makes the request and gets the response by using standard web protocols like SOAP [47], XML-RPC or REST [48]. All of them are protocols based on http and use XML. An XML-RPC call for example could be an http request like this:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
         </param>
      </params>
   </methodCall>
```

and a possible response could be like this:

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
         </param>
      </params>
   </methodResponse>
```

This demonstrates the simplicity and the power of Web Services. All that is required is a web page request and response receipt using the widely supported XML format. Error handling and some primitive data types like integers, strings, structures and arrays are also provided by these protocols.

The second part of the web services functionality is achieved with WSDL [49] and UDDI [50] protocols. UDDI means “Universal Description, Discovery, and Integration” and provides the semantics for describing the needs of an application for a Web Service. That in turn returns a WSDL description of registered Web Services that satisfy these needs. UDDI mechanisms are practically used inside companies to dynamically link web services on a corporate domain.

Summarizing the two above, there are two components that are needed to make a web services interface:

1. A component to pass requests and get responses from a Service Provider
2. A component able to match Web Services to application’s needs

There has been recent attempt to use reconfigurable hardware for web services [51] but obviously web services’ text-based communication protocols are too heavyweight for chip level use.

2.2 Components of a Core Services system

We will now present the way Core Services are realized in a multiprocessor System-on-Chip. What we need is two kinds of hardware and software components as described in the previous section. In this section we will present the general characteristics for Core Services’ mechanics. In order to make Core Services efficient there are many issues which are implementation specific and depend mainly on the communication infrastructure. Details for these issues for common communication schemes can be found in the section 3.3. Core Services don’t define a mechanism for run-time service registration because Service availability is usually known at design time. Ad-hoc service discovery would be an overhead for most SoCs at this moment but is easy to integrate if needed.

The main actors of Core Services are:

The Service Requester: This is a processing element that wants to offload itself or increase performance by performing a Core Service off-chip. A service requester may be running several service requests simultaneously e.g. more than one threads sleeping while waiting for services to complete.

A Service Broker: This is a processing element that manages the assignment of service provider resources to requestors. It is responsible for being up-to date with

system resources availability and allocating them with an efficient way for given performance metrics. More than one requests may be initiated from different requesters and thus an arbitration mechanism is necessary.

A Service Provider: This is a processing element that provides services. It takes the parameters from a service requester processes and sends the response back. Each service provider may provide more than one services but can serve only one request at a time. This eases the design of small embedded systems but doesn't limit larger ones because a single multi-threaded processor may implement more than one Service Providers. Information regarding the capabilities of a Service Provider is being set up on the service broker at design-time but parameters like loading may change at run-time. The service provider doesn't have to retain its state between two subsequent requests.

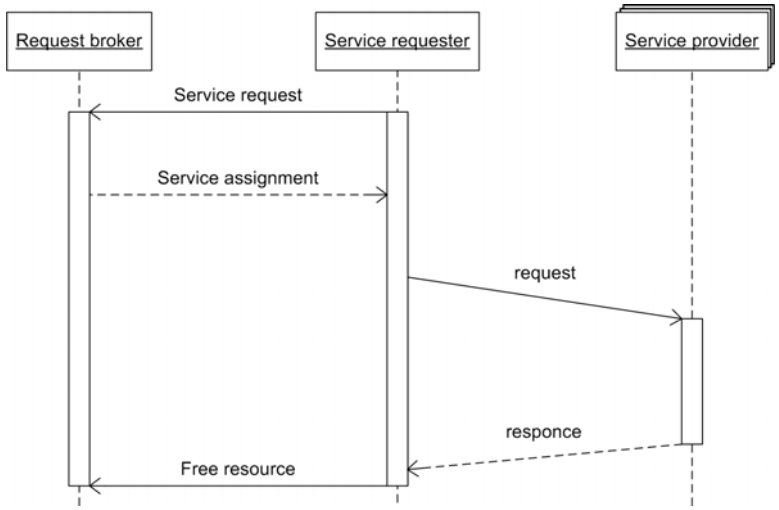


Figure 5. A Core Service transaction

Figure 5 gives an overview of a Core Service transaction. In the middle we can see the Service requester who initializes the transaction. On the left side the object broker analyzes the request and assigns it to an appropriate service provider which we can see in the right side. A transaction is initialized by the service requester and terminates either at the end of service assignment if there is no such service available or at the end of processing by a message to the request broker notifying that the resources are no longer used.

2.3 Advantages of Core Services

Core Services address four main problems of modern System-on-Chip design (see Figure 6):

1. Run-Time Mapping
2. Reconfigurable Hardware Management
3. Fault tolerance
4. Platform based design

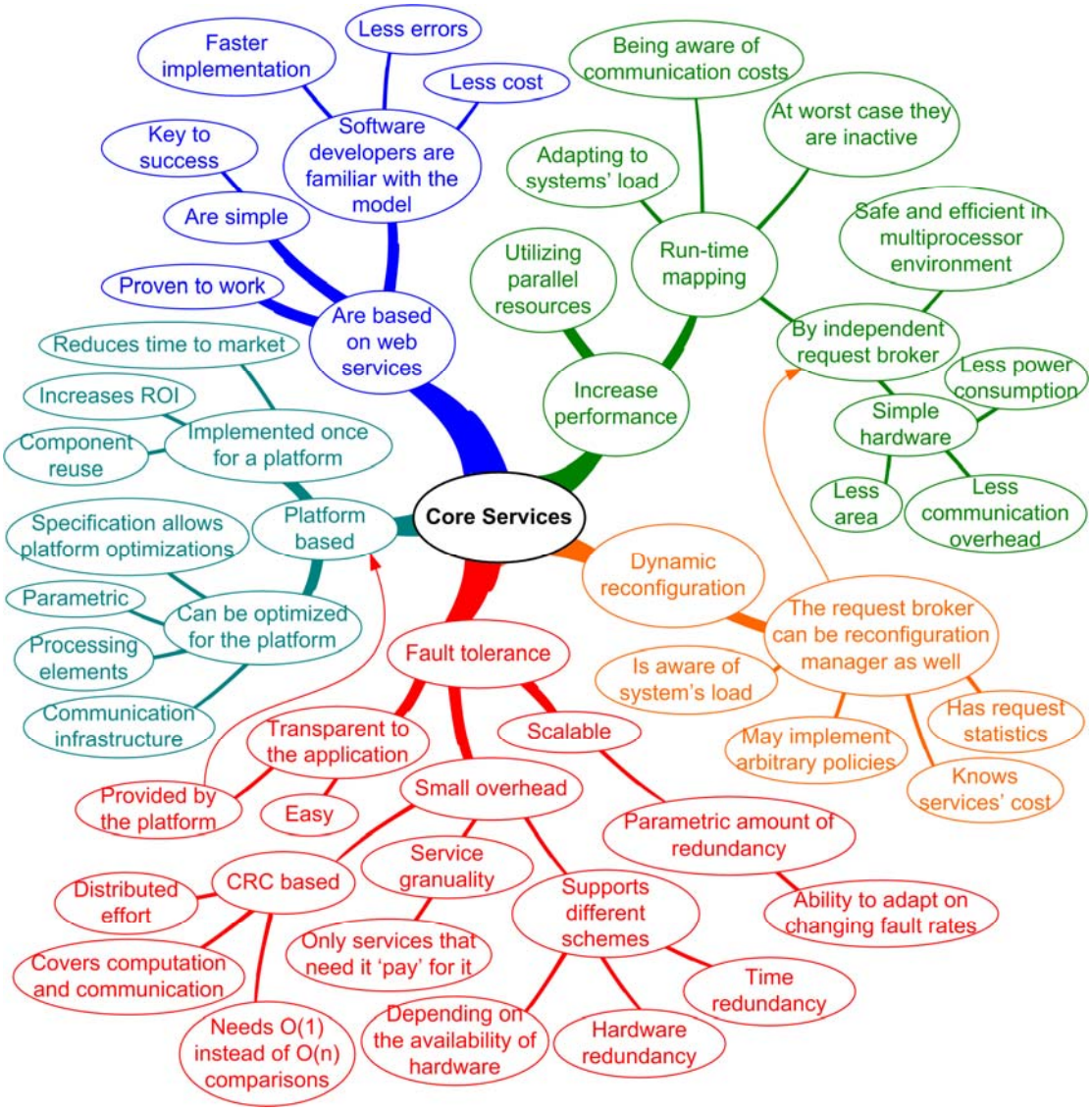


Figure 6. Advantages of Core Services

We will examine these problems, the way that others solve it and what advantages Chip Services provide over other methods in the following sections.

2.3.1 Run-Time Mapping

Traditionally static methods [52, 53] have been used for mapping communication transactions and computation tasks to PEs. In practice [54] mapping distributed applications into NoC architectures is quite difficult even with simple static mapping if the application is not designed for the platform at first place. Many studies have been done in static mapping for minimum energy consumption with realtime constraints by using linear programming [55] evolutionary [56] and other [57-59] algorithms.

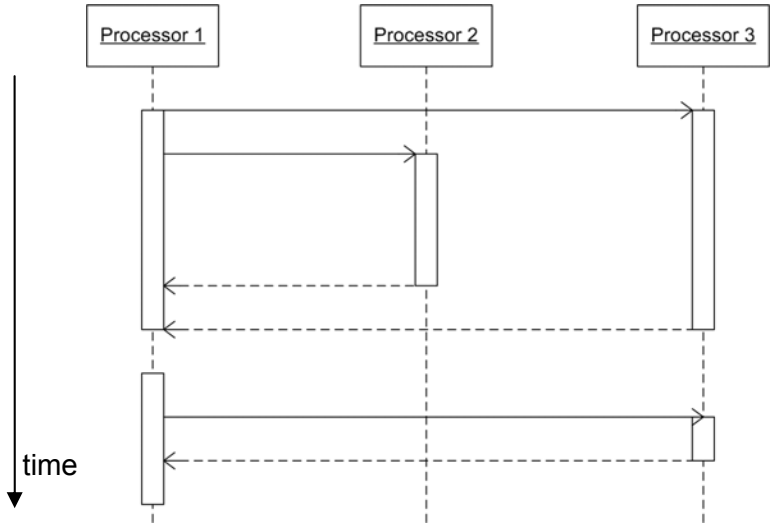


Figure 7. Stating mapping

In Figure 7 we can see an execution scenario with static scheduling. Processor 1 is master and initializes two tasks in slave processors Processor 2 and 3. All of them run in parallel and when they all complete the master Processor 1 returns the function results and is free to start another function e.g. by initializing another task in Processor 3. Stating mapping creates a static schedule on the processing elements of a system. This is being obtained via profiling and defines the performance and the power efficiency of the system.

Run-time mapping has recently started being investigated. In [60, 61] they propose use-cases to reconfigure the network more efficiently, an idea that is also employed in [62]. Generally, reconfiguring the network [63] seems to be the preferred way to reduce network contention and provide fault tolerance [64-66].

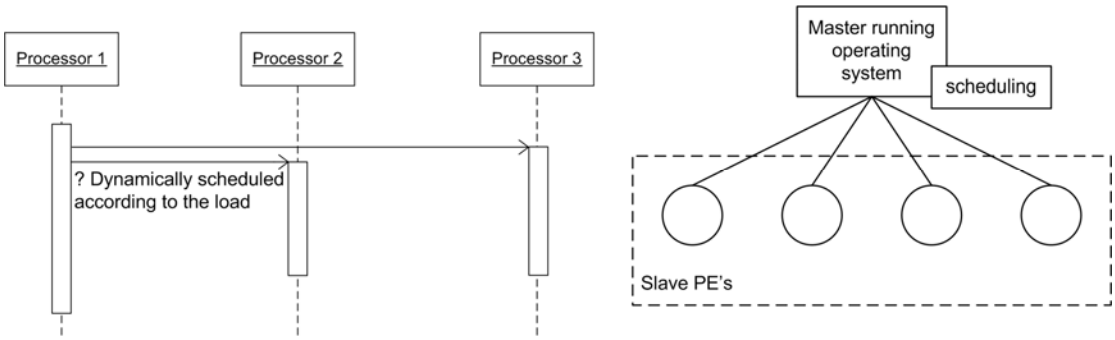


Figure 8. Run-time mapping

As we can see in Figure 8 run-time mapping schedules operations at run time taking into account the computation and communication loads. By scheduling at run-time the system can perform optimally under several different use cases in contrast to static scheduling. As Peng Yang et al. discuss [62] modern standards like e.g. MPEG21 and MPEG4/JPEG2000 execute code based on non-deterministic events and as a result design-time mapping is unable to provide optimum performance. Run time mapping usually employs a scheme where a master processor uses an operating system to profile the system and schedule tasks on slave processors.

Both static and run-time scheduling won't scale well in future MPSoCs because they feature multiple equivalent masters that have to compete for the same accelerating

resources. In order to improve the system's performance we need to accelerate on-demand all the masters.

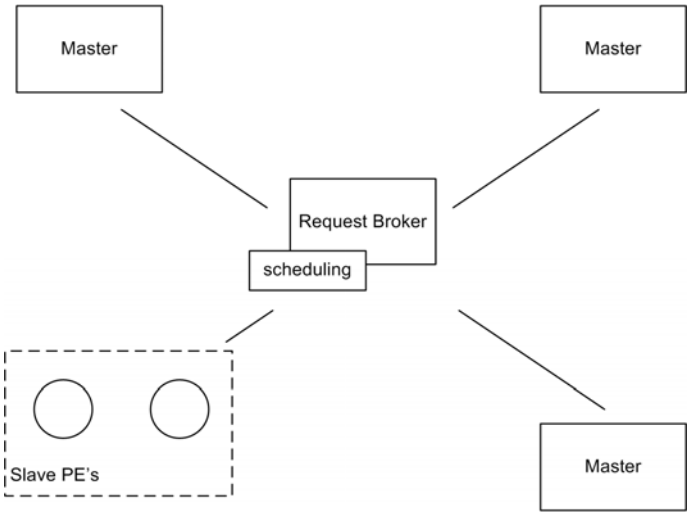


Figure 9. Core Services' mapping

Core Services are designed for these systems. As we can see in Figure 9 the requests from multiple masters are being processed by a centralized scheduler mechanism, the Request Broker. It is trying to optimise system's performance by mapping functionality to Slave PE's (Service Providers). It must be noted that a master processor may also provide services and thus invoke calls into itself.

2.3.2 Reconfigurable Hardware Management

There are two main problems with reconfigurable hardware. The first one is the communication between reconfigurable components and the second one is when to decide reconfiguration. A master device must be able to communicate with the reconfigurable component independently of its current configuration. This requires a level of communication abstraction. Also alternatives of reconfigurable component's functionality must be available in order to address the unavailability of the reconfigurable processing element during reconfiguration. Finally the decision of reconfiguration should be made according to the current processing needs of the application.

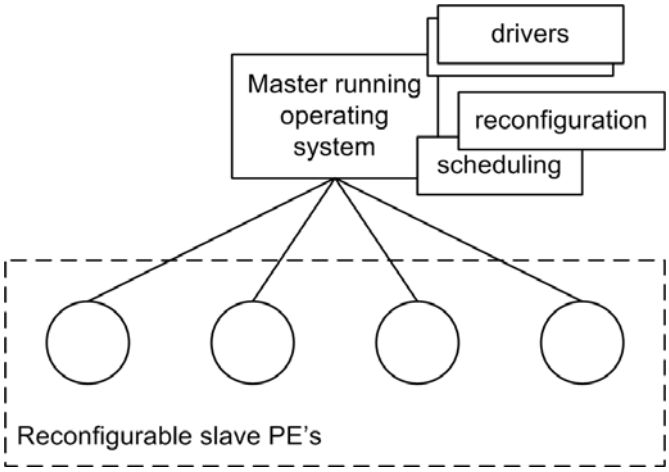


Figure 10. Reconfiguration support at current platforms

All these problems are being addressed with a very Ad-Hoc manner at this moment [30, 67-69]. Communication with reconfigurable components is being done by having several different drivers and using one of them according to the current configuration. The alternative implementation during reconfiguration is usually addressed at application level. Also reconfiguration is being decided based on the demands of a single processor and runs in parallel with the scheduler on the operating system.

Core Services were developed while working on a reconfiguration problem. The problem of abstracting functionality that because of reconfiguration sometimes exists and others not leaded to a run-time mapping solution.

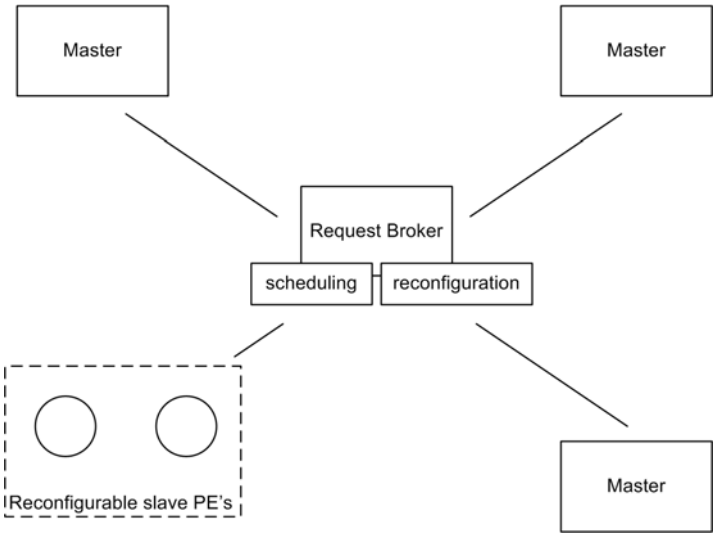


Figure 11. Reconfiguration management with Core Services

As we can see in Figure 11 the request broker has already all the information needed to decide efficiently and transparently if a reconfiguration is needed. It knows the demand for each Core Service and the performance gains if it was available on one or more reconfigurable components. The most important is that by using the Request Broker for reconfiguration management the whole system can be totally unaware of the existence of reconfiguration. It's completely hidden behind the Core Services' Applications Interface (API) and provides optimum hardware acceleration. Operating systems (such as Linux) can be used for Masters without the need of customizations. An additional advantage is that the reconfiguration management can be done in a non application specific manner and thus reused over designs. This is a very important benefit of the Core Services' mechanism.

2.3.3 Fault tolerance

Fault-tolerance allows a system to continue operating properly when some of its components fail [70]. The main problem with VLSI systems is to ensure transient fault-tolerance which means tolerance to Single Event Upsets (SEUs) [71]. Fault tolerance is traditionally being enhanced with the following four methods (Figure 12):

a) Time redundancy: A single function is being executed more than once and the results are compared. This is very popular technique since it requires less hardware/software resources but a multiple of the original time of the function is being used decreasing performance.

b) Hardware redundancy. In contrast to time redundancy, this technique uses a large number of modules each one executing the same function and their results are being

compared. Obviously this requires a lot of extra resources but it can achieve increased performance.

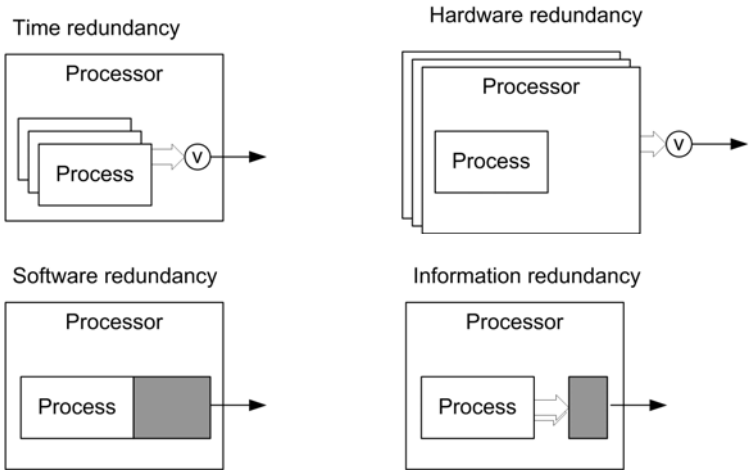


Figure 12. Fault tolerance methods

c) Software redundancy: In this approach sanity tests are being applied to the output data in order to verify that they are correct. This requires extra time but less than time redundancy technique. The fault tolerance techniques of this kind are highly application dependent and can hardly get reused.

d) With information redundancy additional information is being used e.g. checksums that verify that data are correct with less performance penalty than time and software redundancy.

The main problem of fault tolerance is adjusting the number and the kind of redundant system resources. We have again a resource management problem. Partial reconfiguration can help in the correction of SEUs on the configuration memory as we can see in [72]. We use it to correct efficiently SEUs on application-level functionality. Only a small amount of system's processes are critical enough to need fault tolerance.

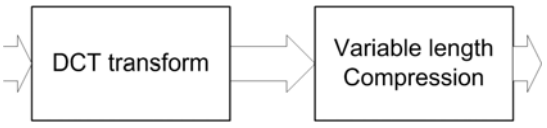


Figure 13. Typical audio/ image commercial application

For example in commercial applications we frequently find computational requirements for data-oriented algorithms e.g. DCT and control oriented e.g. compression algorithms as shown in Figure 13. A fault in the DCT algorithm is insignificant but a fault on the compression algorithm is critical since it may cause loss of synchronization and system failure. If both the DCT and the compression algorithms can be accelerated on a reconfigurable hardware component which one should get accelerated? The answer is not straightforward since the two algorithms share a common data stream. If we accelerate the DCT algorithm and choose to have time redundancy for compression we may end up with a bandwidth that the latter can't handle. If we provide hardware fault tolerance to the compression algorithm the DSP algorithm may be unable to provide the appropriate bandwidth to the compression algorithm. Carefully tuned dynamic resource management switching

between hardware and time redundancy gives the best solution in this scenario and most real-life applications are like this.

Core Services use the run-time mapping mechanism to provide fault tolerance with an optimum way at a given time. The most important advantage is that they achieve this without any development effort. Fault tolerance is being provided by the framework and can also be used at design time for debugging purposes i.e. verifying the equivalence between hardware and software implementations. Each service call is being invoked with a redundancy parameter which specifies how many voting results (up to 16) should agree to accept a result. These voting results are being obtained by executing the service in the fastest combination of (hardware or software) service providers available at that time as we can see in Figure 14.

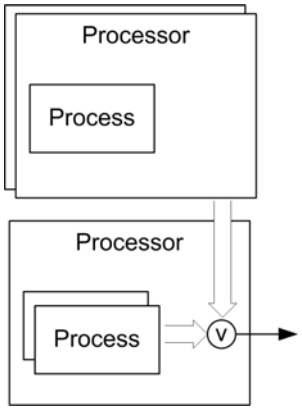


Figure 14. Fault Tolerance with Core Services

The results of a Service may be large vectors and comparing them would be a waste of time but most importantly waste of precious bus's cycles in order to collect them in the core where voting takes place. In Core Services only one Provider returns the full set of results and a checksum while the others return only their checksums saving significant bus bandwidth. The kind of checksum is not specified by Core Services and this allows significant application specific optimizations. In a software dominated platform the use of a Linear Feedback Shift Register (LFSR) for checksums would require significant computational resources. Additionally by comparing only the checksum we may be able to save time by not performing operations that produce invariant results in respect to the checksum. For example in many encryption algorithms the final step is a scrabbling of the output parameters. If the checksum is a simple addition this last step is unnecessary because the sum of numbers is invariant to their order.

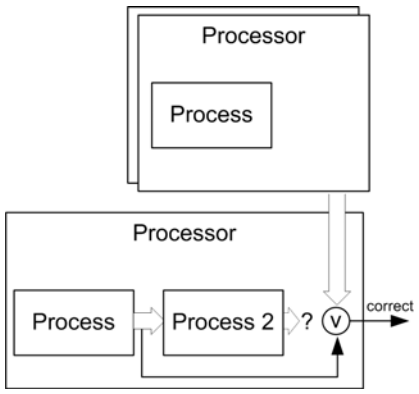


Figure 15. Out of order fault tolerance with Core Services

Finally the processor may go on with its computations as soon as the first full result arrives assuming that it's correct. Checksum comparison may take place latter when all Services complete and in most of the cases it will be correct giving near non-fault tolerance performance (see Figure 15).

2.3.4 Platform based design

Platform based design promises to increase productivity, decrease time to market and development costs by re-using out-of the self pre-verified components within a platform framework [73]. At this moment platform based design as provided for example by Xilinx via Xilinx Platform Studio (EDK) is a good step towards these promises. You can easily develop a complex system consisting of IP cores provided by Xilinx and its partners. For example for the XUP Virtex II Pro platform (see Appendix A) one could easily connect two PowerPC cores with RAMs and other peripherals by using the CoreConnect hierarchical bus. It's quite simple to create the hardware and basic software drivers for this platform by using XPS giving a kick-start for developing new applications (see Figure 16). There is also the option to use MontaVista linux which offers all the software friendliness of Linux and device drivers for most of the essential peripherals of the platform.

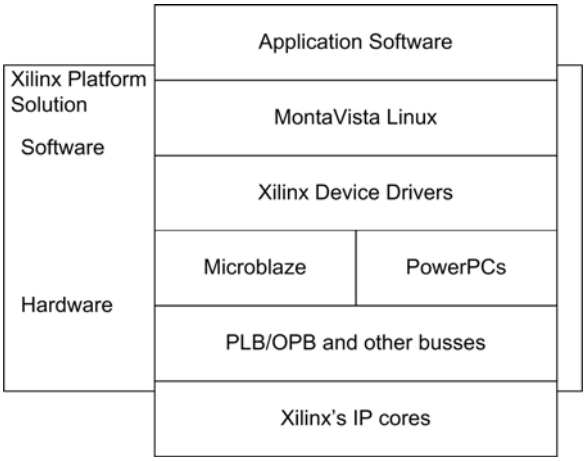


Figure 16. Platform based design with Xilinx Platform Studio and MontaVista linux

Still, the question remains: Why to use Xilinx's Platform Studio when there exist more efficient processors with equivalently large set of peripherals and at least equally good software support in a less power hungry and cheaper application oriented System-on-Chip like Philips' Nexperia. The answer is reconfigurability. Xilinx's flow supports the addition of custom hardware accelerators who offer the potential of unbeatable acceleration.

But how well does this platform's flow supports its only competitive advantage? The answer is not so well. There is only one wizard that allows the creation of a template custom hardware and software peripheral. In fact a developer for this platform can't avoid studying the underlying PLB/OPB busses, work explicitly to convert the endianess's of the signals and resolve a lot of handshaking problems. From software side a template is provided that slightly abstracts the underlying bus. A custom driver has to be developed in order to transfer data to the custom core and get responses back. Of course there is no support at all for Linux for which someone has to re-develop a complete device driver in order to access the peripheral. After all this effort what does someone have? A component that is very tightly coupled to the platform. If the communication infrastructure changes e.g. turn to NoC then the hardware will

have to be modified again or at least get “wrapped” possibly loosing some of its performance. The device drivers also will change. Hopefully if the device driver is well written, user applications won’t need modifications. The worst of all from a business point of view is that all these IPs are very platform specific and for example for sophisticated components it would be very expensive to switch from e.g. Xilinx to Altera. Obviously every company wants to be vendor independent in order to take advantage of vendor’s competition so it’s forced to implement its own abstraction over these platforms for its IPs.

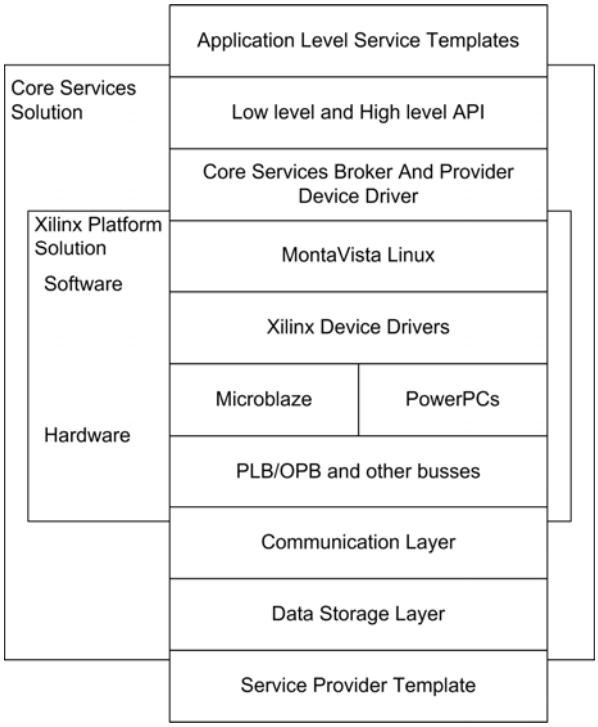


Figure 17. Core Services' stack over Xilinx’s Platform

Web Services deal with machine-to-machine interoperability and Core Services do the same on a platform-to-platform level. They achieve this by adding one software and two hardware abstraction layers over Xilinx’s flow. From the hardware layer the communication layer hides completely the bus and Core Services’ communication protocol and provides an interface suitable for memory-like (slaves) components. On top of it that the data storage layer holds the data in the form of one independent RAM for each variable used for a service provider. It provides an interface that is suitable for processor-like components (masters). On top of it lies the actual Service Provider which can access each variable independently increasing the maximum achievable throughput and thus minimizing the computational time. By having two layers of abstraction we support two stage reconfiguration [29]. Implementation at the top abstraction level is very simple. For example for the calculation of a mathematical expression like $\overrightarrow{out} = \overrightarrow{in1} + \overrightarrow{in2} \cdot \overrightarrow{const}$ an implementation could be almost as simple as this:

```
out[count] <= in1[i] + in2[count] * const[count];
outwe[count] <=’1’;
```

Passing the variables, returning the parameters and communicating with the software is being handled by the middle layers.

From software side there are similar improvements over the Xilinx's platform. A Linux Device Driver that provides access to the Service Broker and the Service Providers is provided. Each of them is being accessible as a single device in the /dev/ directory. On top of that the low level and high level API are provided to the applications. The low level API provides more flexibility to the application developer in terms of performance but requires better knowledge of the Core Services' functionality. The high level API uses the low level API and gives the developer a completely abstract remote procedure call interface. No knowledge of Core Services internals is required at this level. For example a call using the high level API would be as simple as this:

```
void funDefaultMAC(int * in1, int * in2, int* out) {
      // Default (software) implementation
      for (int i = 0; i < 10; i++)
            out[i] = in1[i] + in2[i] * const[i];
}
…
csMAC(funDefaultMAC, in1, in2, out, NO_REDUNDANCY);
```

As you can see, there are no hardware references at all. This abstraction level is much more platform independent than a traditional call like this:

```
send_peripheral_data(REQUEST, BROKER_ADDRESS);
p = get_peripheral_data(BROKER_ADDRESS);

for (int i = 0; i < 10; i++)
   send_peripheral_data(in1[i], PERIPHERAL_BASE_ADDRESS[p]);
…
```

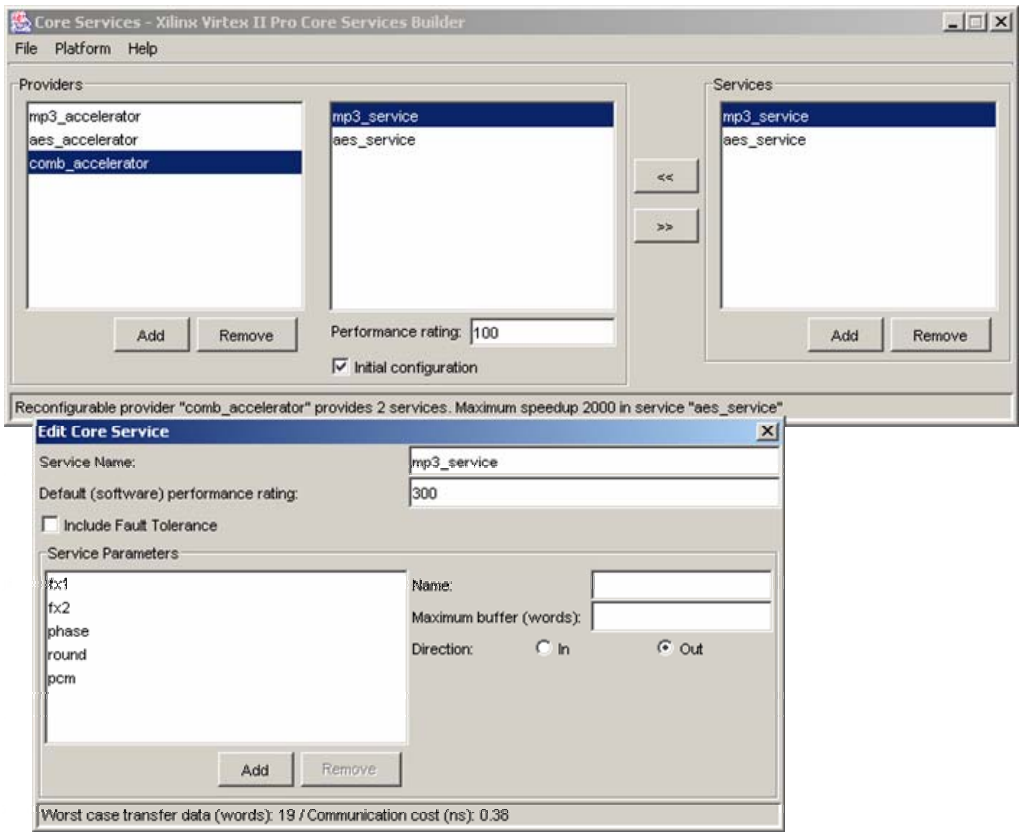


Figure 18. Core Serices Builder: The Core Services' GUI

As described in [73] with the ever-increasing pressure of time to market infrastructures and tools must be developed in synchrony with design methodology. Core Services' methodology is being accompanied by tools and infrastructure including Core Services Builder (see Figure 18), a user friendly interface via which one can customize Core Services and Service providers and have all the hardware and software created automatically. The tool provides interactive information for the communication costs and performance gains in order to aid designer's decisions.

2.4 Related Work

Several other researches have attempted to solve some of these problems with approaches similar to Core Services. We will present the most similar works and highlight the points where Core Services differ.

Object oriented methodology has the idea of polymorphism which means that a function call may map to different implementations according to the class of the object that it belongs to. In [74] they expand their previous [75] ASIP (application-specific instruction set processor) methodology to use the NoC to dispatch with zero overhead virtual methods to hardware or software implementations. They do not provide dynamic resource management as Core Services.

In [76] they present a Java based abstract stream decoder technology for reconfigurable hardware. They lack many of the strong semantics of Core Services. In the very interesting and somehow technical [77] transparent management of reconfigurable hardware or software components is being proposed. They use CAN bus for communication. "Run-time optimizations" and "partial run-time reconfigurable modules" are left for "future versions".

The work in [78] has many similarities with our by referring to Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI) and having an "IP Core Lookup Service" which instantiates objects and decides reconfigurations which is similar to our object broker. They also use objects to abstract hardware which is not quite suitable for reconfigurable hardware and it naturally leads them to use ad-hoc methods for serialization in order to aid relocation. Although they claim that their methodology improves performance they have no mechanism to guarantee it as they don't consider computation or communication costs at all. They assume hardware implementations are faster than software without considering the communication costs and they implicitly assume that all hardware accelerators provide the same speedup for a functionality which is true only if the same implementation is being used. Core services are stateless which simplifies both hardware and software and makes relocation unnecessary. They also provide reconfiguration mechanism and run-time mapping which optimize for performance and guarantee acceleration via the performance deadline.

On the other hand in [79] they present a heuristic similar to ours (see section 3.2.3) for computation and communication costs and they use it to map tasks to PEs. All the resource management is being handled by a single master processor and although the technique is supposed to target on multi-processor systems they describe a one-master-many-slaves architecture. Core Services Phase I mechanism (see section 3.2.1) allows multiple masters to gain access to accelerating resources.

Chapter 3. Core Services methodology, mechanics and implementation

3.1 The methodology

Core Services are meant to be the equivalent of web services optimized for on chip communication. The steps of the Core Services methodology are the following:

Step 1. Profile the system and sort its functions by the total amount of time spent on each.
By starting this methodology it is assumed that there exists a functional software prototype of the system. By profiling we identify the functions that consume most of the time. Obviously this is the first place to look optimizing system's performance.

Step 2. Decide if they are suitable for hardware implementation.
We give a detailed description on section 3.4 on how to decide if a function is suitable for hardware implementation.

Step 3. Replace with service calls and provide default service implementation.
Assuming that the platform has a Core Services' implementation this is as easy as modifying slightly the original function and calling it via Core Services' wrapper.

Step 4. Test the software only implementation on the platform.
At this level we can verify that the system works correctly after adding the Core Services' functionality by using the default software service implementations.

Step 5. Calculate the estimated savings of making a hardware accelerator for this function. If constrains aren't yet met, accelerate more functions.
Communication cost for the given platform can be accurately estimated at this stage because the length of the call and return values are known. A hardware engineer can give estimates on the speed of a hardware implementation of a certain functionality. As a result we can know in such an early stage if the savings suffice to meet system's constraints. If not we can turn more functions to Core Services. If hardware implementations can't or needn't be provided latter the system will still be functional.

Step 6. Create hardware test data.
By running the application as described on the previous step, we can create as many test data as required with an automatic or semi-automatic way. Time consuming system-level simulations are being avoided by using Core Services.

Step 7. Create hardware instance of Core Service using automatically generated service stack and verify with the test data.
Hardware templates generated by Core Services' fingerprint and test data can significantly ease the implementation of the hardware component. Manual optimizations on the templates can be performed if time allows.

Step 8. Calculate the actual savings of making a function Core Service. If the constrains aren't yet met, create more hardware components or optimize more the existing ones.
At this stage we have actual data for the time it takes the hardware to complete the service. Precise evaluation of the savings can be performed.

Step 9. Test the software/hardware implementation on the platform.

Now system level testing can be performed. Testing can be eased by the fault tolerance facilities of Core Services. Both software and hardware instances are being called and if the checksums of the results aren't the same warnings may be issued.

3.2 Core Services' communication protocol and algorithms

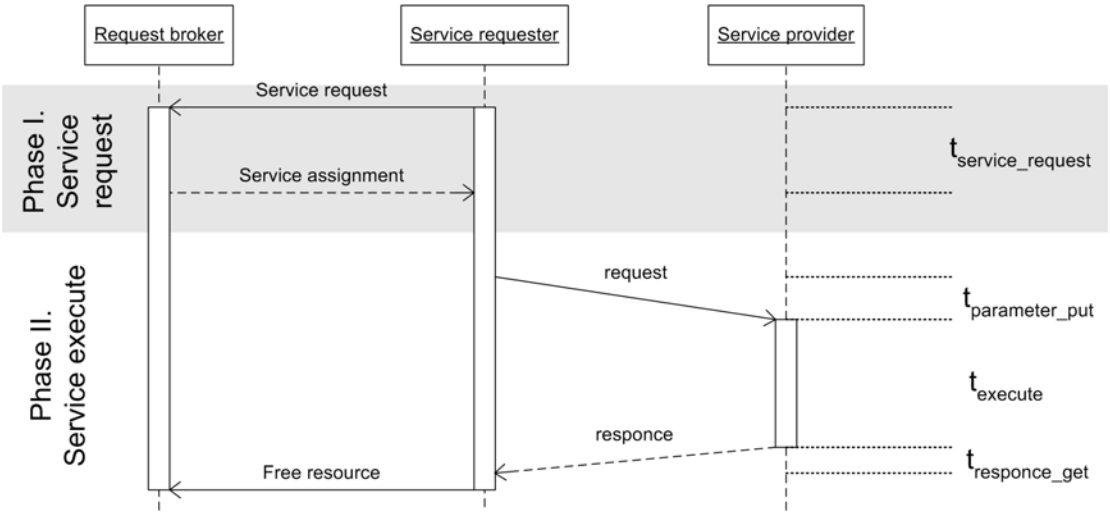


Figure 19. Core Service transaction details

Figure 19 shows a Core Service transaction. We will present the details of the communication protocols and the algorithms that are involved in a Core Service transaction.

3.2.1 Phase I. Service request

3.2.1.1 Service Request

Phase I of a service transaction is the service request. In this phase, the requester contacts a request broker and asks for service. The form of a request packet can be seen in Figure 20.

0 — 15	16 — 31		
Service ID	Requester ID	Providers	0001
Performance deadline			

Figure 20. Service request packet format

The MSB of the first word is 1 to denote a service request packet. The fields on a service request packet are the following:

1. The service ID of the service requested
Each service is described by its unique ID. This is system-level unique and is assigned at design time. It is 16-bits long giving a maximum of 65536 services.

2. The ID of the service requestor
The ID of the service requestor is needed in order to return the response to the requester and calculate inter-core costs. For example the communication costs over NoCs may depend on the distance between the cores. This is 8-bit field so it supports up to 256 service requesters.

3. The number of service providers requested
The number of service providers may seem unnecessary but its' necessary to support fault tolerance. A single request may require more than one service providers for example two for fault tolerance in a moderate faulty environment. 4-bits are reserved for this number giving a potential of 16 service providers per request. If an application that requires fault tolerance is run on a platform that doesn't support it the performance will be reduced compared to another platform that supports it because of software simulation of multiple calls.

4. The performance deadline to be met
The performance deadline is the number of clock cycles that the service requester estimates it will take itself to complete this service. This is being used to provide a run-time adjusted threshold to the service broker in order to decide if it's reasonable to assign a service provider to this request.

In fact the performance deadline is just a metric and it isn't bound only to time. It could be for example a combination of energy and time. In most cases, better time performance means also better energy performance [80-82] so it's reasonable to use time as a metric. The easiest way to statically predict this number is by profiling. If a big variance in execution time is expected, the service requestor should multiply it by a load factor dependent, for example, on the number of active threads/processes in order to reflect its load. In might also be useful to under-estimate slightly if it's more important to off-load the requester in order e.g. to provide better behaviour in unpredictable real-time events.

3.2.1.2 Service assignment

The response by the service broker can be seen in Figure 21.

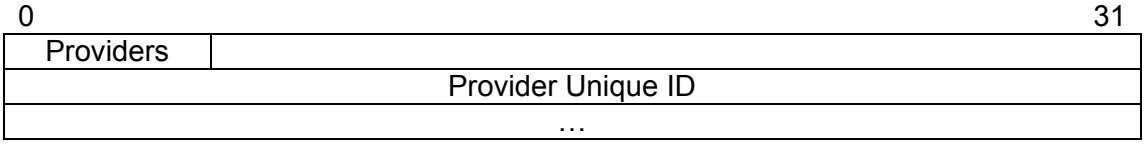


Figure 21. Service assignment packet format

The fields on a service assignment packet are the following:

1. Number of providers
This is the number of providers that have been allocated to the requester for this request. It may be less than the number of service providers requested e.g. zero if no such service is available or none meets the performance deadline. This field is 4-bits long as the respective request field.

2. Provider ID
A list with unique identifiers for the specific service follows. These ID's are being used to send the parameters as described bellow. The meaning of these ID's is platform depended and may be for example base addresses, IPs or port numbers.

3.2.1.3 Why is a service broker needed?

A service broker is needed at least as a means of having a central repository of services and their availability status. An alternative to this would be to have a completely ad-hoc system in which each request would be broadcasted to the 'market' of service providers, they should respond by bidding for the service request and the best offers would be accepted. No matter how nice this scenario sounds it

has important performance and power drawbacks. The service request gets broadcasted to all the service providers, even those that may not provide a service. Additionally a large number of bids would have to be communicated even from providers that have non chance of being accepted. Obviously all these end up in a waste of performance and power. The existence of the central 'marketplace' of the service broker saves all these resources.

3.2.1.4 Why does a service broker make the service assignment?

Alternatively the request provider could be given a complete list of services and metrics let decide itself. Apart from the increased interface complexity and the communication overhead such an implementation there are other important drawbacks. The external broker DEDICATES the providers to the requesters. If this dedications didn't hold true in a multi-threaded but even more in a multi-processor environment a lot of collisions would occur. Requesters of the same Core Service would reasonably try to allocate the best resource. The first one checks if it's free and locks it with an atomic operation. The others check with failure and either look for an alternative service provider or wait until the first one completes and take their turn. Obviously this is too much overhead for an embedded system. It is much better to have a central locking mechanism that manages the service providers. This is true for small number of service requesters. If there were a lot of service requesters the availability of the broker would become an important bottleneck. However a service broker can easily be implemented by multiple cores using a shared memory for their synchronization. Obviously the decision of making the assignment on the service broker scales well even in much larger systems.

3.2.2 Phase II. Service execute

3.2.2.1 Parameter passing

Phase II of a service transaction is the service execution. In this phase, the requester communicates directly with service provider to pass the parameters and get the response.

The fact that the service provider doesn't have to retain its state between two subsequent accesses simplifies both hardware and software. We can't be guaranteed that two subsequent service requests are going to be serviced by the same service provider. As a result, the state should be transferred from one service provider to another which is generally impossible because different service providers may be incompatible of each other. Even in systems where only one service provider exists for each service a request from another thread may interleave between two subsequent requests from a thread resulting in service corruption.

It may seem as a waste of scarce system resources that we write data from the main memory to the hardware accelerator and we should prefer a shared memory model where data are being passed by reference instead. Our decision is safer for several reasons:

1. Data may not be in the main memory but in a local cache.
2. Even if you move data by reference, the service provider still will have to read them, so you don't save the reading part.
3. Core Services give that ability implicitly. One can pass a pointer as variable and a service provider capable of reading the main memory will use it to read the data. In this case one must be careful a) to give a valid physical memory location which may be different from a pointer's virtual address b) to resolve cache coherence and memory ownership issues that may arise.

The ability to have variable sized variables is important in order to support functions with strings as parameters.

Core Services' parameters and responses are only arrays of 32bit words. This is done because it's only meaningful to run a service on another processor if it takes considerable amount of time. Apart from random number generation and some rare cryptographic applications with extremely high complexity large amount of time means large amount of data as well.

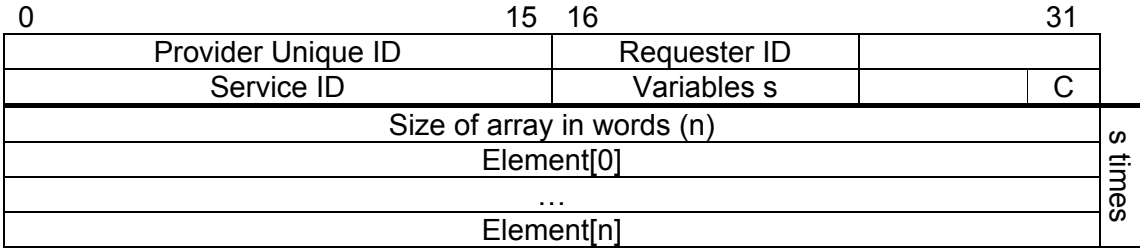


Figure 22. Request packet format

The format of a request packet can be seen in Figure 22. It contains provider unique ID, the Requester ID that will be used to receive the response, the service ID requested and the number of variables (parameters) that are being passed to the service. The (C)hecksum flag tells the provider if a full reply should be send or just the checksum for each variable. Then s parameters follow each one prefixed by its size in words and then the words of the data in sequence. At the end of this packet, the computation begins.

3.2.2.2 Response getting

The service provider sends back the response. This may be done by pulling or by pushing depending on the platform i.e. the requester may read the data from the provider or the provider may send the data to the requester. In the former case, there must be a mechanism to tell the requester that the computation was completed and the data are ready. This may be done by polling or more efficiently by an interrupt if the platform supports it.

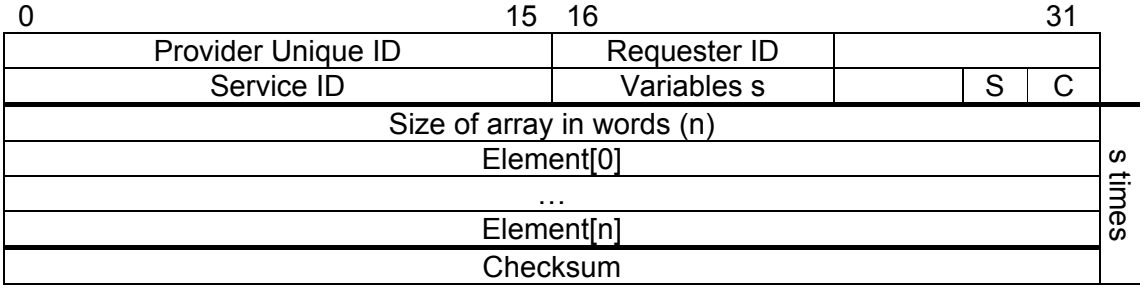


Figure 23 Response packet format

A similar format with the request packet is being used for response packets as shown in Figure 23. The only difference is the existence of a checksum that is being used if the service supports fault tolerance. If the S flag is set, then the packet contains checksums. If the C flag is set, then the packet contains only the checksum and not the variables themselves. Bare packet's size and thus communication cost is significantly reduced. The kind of checksum is platform dependent and may be more or less software-friendly.

3.2.2.3 Free resource

A free resource packet (see Figure 24) is being sent to the service broker in order to notify that the providers have been freed and can get reused.

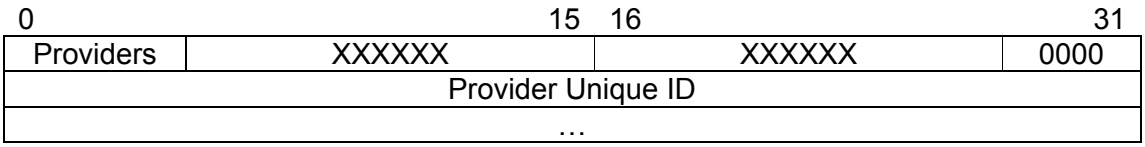


Figure 24. Free resource packet format

The MSB of the first word is 0 to denote a free resource packet. The fields on a free resource packet have similar meaning to the ones of resource assignment packet (see section 3.2.1.2).

3.2.3 Mapping algorithm

Mapping of the requests to the service providers is being implemented by the object broker. There are many different ways to do them but the more intuitive way is the following one that is actually being suggested by the Core Services' infrastructure.

A request provides the following data: The number of providers requested n_p, the deadline t_d, the service ID S and the requester ID ID_r. We can see in Figure 19 four times that characterize a transaction: $t_{service\_request}$, $t_{parameter\_put}$, $t_{response\_get}$, $t_{execute}$. The $t_{parameter\_put}$ and $t_{response\_get}$ can be merged to a single value, the communication time: $t_{comm} = t_{parameter\_put} + t_{response\_get}$.

The problem is that we have a set of service providers SP from which only $ASP \subseteq SP$ are available at a given time and each providing a set of available services A_{sp} and we want to return a subset $X \subseteq ASP$ of $a_p \le n_p$ service providers that provide the service $S \in A_x, \forall x \in X$, have minimum cost $c(x) \le c(s), \forall x \in X, \forall \{y \in SP \mid y \notin X\}$ and less than the performance deadline $c(x) < t_d, \forall x \in X$. We would like to have a fast algorithm because it affects the $t_{service\_request}$ which is a pure overhead in service call.

The easiest way to find the best X is to maintain a list with one node for each available service provider which has a list with one node for each available service A_{sp}. This algorithm has a worst case complexity of $O\left(size(ASP) \cdot max\left(size\left(A_{sp}\right)\right)\right)$ which may be bad for large number of service providers or available services.

```
bestcost ← ∅
X ← ∅
for each service provider x in ASP do
      for each service s in A_x do
            if ( s = S and c(x) < t_d and c(x) < bestcost(t), ∀t ∉ X ) then
                  if ( size(X) > n_p ) then
```

```
                        find_and_remove(worst_element( X ))
                    end if
                    bestcost ← bestcost ∪ c(x)
                    X ← X ∪ x
                end if
            done
done
```

Another algorithm which has much better performance can be implemented by using hash tables of services h_s with each entry containing a sorted list of active service providers l_{ASP} for that service. This is a very fast algorithm that gives very fast access time $O(1)$ but requires much more memory and considerable housekeeping effort to keep the structure up to date when availability of services changes.

$$X \leftarrow h_s[S][1...n_p]$$

Both algorithms are useful and there are a lot more algorithms in the middle between them. The first one is good for hardware implementation or in a processor with limited amount of memory or in a system that has a small amount of service providers and services per provider. The second one is more suited to large systems with many service providers and services and with taught deadlines that can't afford an increased $t_{service\_request}$ but can afford the extra memory needed to keep a well organized registry.

The main problem that we set aside until now is the calculation of the cost for executing a Core Service in a service provider $c(x)$. There are two dangers. The one is to oversimplify and reduce the efficiency of run-time mapping, the other is to over-analyze and reduce the performance benefits because of the computational overhead of calculating the cost. We want to include the computational and the communication costs.

The computational costs can be represented as a $n_s \times n_p$ matrix c_c of the average clock cycles of execution where n_s is the number of services, n_p is the number of providers. Run-time variable load factors can be expressed by a vector lf of n_p elements. $lf(i)$ = 1 when the processor i is average loaded, < 1 if it's under-loaded and > 1 if it's overloaded. The computational cost including loading is $c_{comp} = c_c \cdot diag(lf)$.

The communication cost can be approximated with an average bandwidth $bw_{i,j}$ in clock cycles/word from service requester i to service provider j. The number of words per service w_s includes parameter passing and response getting and is known at design time. The total communication cost $c_{comm} = w_s \cdot bw_{i,j}$ is known at design time and can be represented with a $n_s \times n_r \times n_p$ matrix. Note that if the communication costs present large variance and the average bandwidth $bw_{i,j}$

induces large errors we can easily add a run-time variable load factor to this model as in the case of computational costs.

The total cost $c(x \mid r,s,lf)$ for a given service s, requester r and load factor lf, can be calculated with three array accesses and a multiplication. It is $c(x \mid r,s,lf) = (c_{comp})_{s,x} + (c_{comm})_{s,x,r}$. If these two matrixes c_c and c_{comm} are set, Core Services are ready to work.

For example in a system with 2 service requesters and 3 service providers for 2 services. Given a run-time load factor $lf = (1.1 \quad 1 \quad 0.8)$ and a computational cost matrix

$$c_c = \begin{bmatrix} 30 & 20 & 70 \\ 40 & 50 & 12 \end{bmatrix}$$

we have a total computational cost of

$$c_{comp} = c_c \cdot diag(lf) = \begin{bmatrix} 33 & 20 & 56 \\ 44 & 50 & 9.6 \end{bmatrix}$$

For given inter-processor bandwidth $bw_{i,j} = \begin{bmatrix} 1.3 & 1.3 & 2 \\ 1 & 1 & 1.3 \end{bmatrix}$ and $w_s = (10 \quad 20)$ words per service we have a communication matrix:

$$c_{comm} = \left(\begin{bmatrix} 13 & 13 & 20 \\ 10 & 10 & 13 \end{bmatrix}, \begin{bmatrix} 26 & 26 & 40 \\ 20 & 20 & 26 \end{bmatrix} \right)$$

The total cost of services is:

$$c(x \mid r,s,lf) = (c_{comp})_{s,x} + (c_{comm})_{s,x,r} = \left(\begin{bmatrix} 46 & 33 & 76 \\ 43 & 30 & 69 \end{bmatrix}, \begin{bmatrix} 70 & 76 & 49.6 \\ 64 & 70 & 35.6 \end{bmatrix} \right)$$

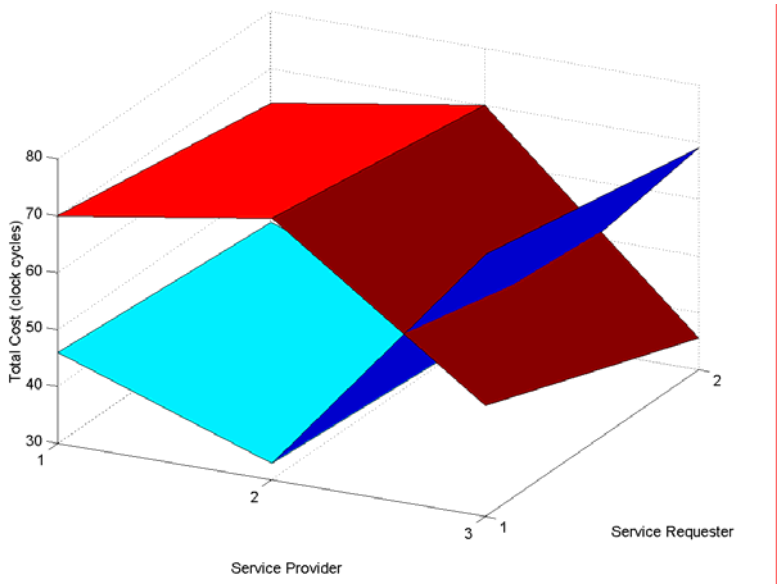


Figure 25. Total cost for two Core Services

As we can see in Figure 25 the cost of the two Core Services varies not only with the service provider but also with the service requester.

3.2.4 Reconfiguration management

Given that we have a reconfigurable service provider able to provide n services $S_1 ... S_n$ and currently it's configured to provide service S_c. Each service gives a corresponding speedup of $s_1 ... s_n$ each time it executes and in the last $frame$ requests we have $n_1 ... n_n$ requests for each service. The reconfiguration penalty is t_p during which it can't provide any service.

The potential speedups in the last $frame$ requests for the i-th service are $s_i \cdot n_i$. There is a best service S_b for the last frame for which $s_b \cdot n_b \geq s_i \cdot n_i, \forall i \neq b$. If there are more than one best services with equal potential speedups we select as S_b the current service S_c if it belongs to that set otherwise a random service from that set and we continue.

If the best service S_b is the one that we already run S_c then there is nothing to do. We have an optimum solution. If not then we have to examine the opportunity cost, "the most valuable forgone alternative". That's $s_b \cdot n_b$, the potential speedup for our best service. But we are already having a profit of $s_c \cdot n_c$ because of the chosen service S_c. So the actual loss is: $t_{loss} = s_b \cdot n_b - s_c \cdot n_c$ and it has units of time. Assuming that we will have the same profile of requests in the near future, If we invest t_p time of inactivity we will gain a future profit increase of $t_{loss} = s_b \cdot n_b - s_c \cdot n_c$ per t_{frame}. We just have to define a threshold time t_{th} in the order of t_p. If the accumulated t_{loss} for the same S_b exceeds t_{th} we decide a reconfiguration $S_c \leftarrow S_b$. The algorithm has a complexity of $O(n)$ because of argmax and is as follows:

```
After a frame is complete:
S_b ← argmax(s_i · n_i)
t_loss ← s_b · n_b − s_c · n_c
if ( t_loss > 0 ) then
      if ( S_b ≠ S_blast ) then
            t_cut_loss ← t_th
      else
            t_cut_loss ← t_cut_loss − t_loss
            if ( t_cut_loss < 0 ) then
                  Reconfigure to S_b
            end if
      end if
      S_blast ← S_b
end if
```

For example if we have two services S_1 and S_2 that provide a speedup of $s_1 = 60\mu s$ and $s_2 = 300\mu s$ respectively and the chosen service at given time is $S_c = S_1$. From the statistics we know that in the last $frame = 20$ requests that lasted $t_{frame} = 1ms$ we have $n_1 = 15$ and $n_2 = 5$ requests. The reconfiguration penalty $t_p = 10ms$ and thus we define a threshold time $t_{th} = 5ms$.

Service:	S_1	S_2
Potential speedup	$s_1 \cdot n_1 = 900ns$	$s_1 \cdot n_1 = 1500ns$

Obviously the best service $S_b = S_2 \neq S_c$. The $t_{loss} = s_b \cdot n_b - s_c \cdot n_c = 600\mu s$. If S_2's potential speedup continues to exceed S_1's for subsequent frames with the same rate, then a reconfiguration will be decided after $t_{th}/t_{loss} = 5ms/0.6ms = 9$ frames = 180 requests.

To demonstrate the operation of the algorithm we will do simulation. We model each service as independent Markov Chain that has two states Active (A) and Sleep (S) and transition probability p_A and p_S respectively (see Figure 26).

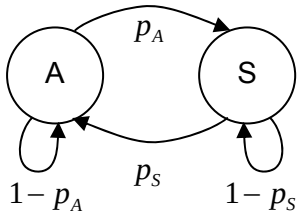


Figure 26. A service as a two state Markov process

With this model we create simulation events and we check the behaviour of the reconfiguration algorithm. More specifically we can model the following kinds of services:

		p_S	p_A	T_S	T_A
a)	infrequent that last a lot	0.1	0.1	50%	50%
b)	frequent that last a lot	0.4	0.1	40%	60%
c)	infrequent that don't last a lot	0.1	0.4	60%	40%
d)	frequent that don't last a lot	0.4	0.4	50%	50%

Examples of these categories are the following:

	A program that uses a service:	Example sequence
a)	heavily in a single routine	SSSSS**AAAAAA**SSSSSSSSSSS**AAAAA**SSSS
b)	heavily on multiple threads	S**AAA**SS**AAAAAA**SSS**AAAAAA**SS**AAAAAAA**SS
c)	occasionally in a single routine	SSSSSSSS**AA**SSSSSS**AA**SSSSS**AA**SSSSSSS
d)	occasionally on multiple threads	S**A**S**A**SSSS**A**S**A**SSS**A**SS**A**SSAS**AA**SS**AA**SS**A**S

T_S and T_A in the table above are the amount of time spent in sleep and active state respectively as predicted by Marcov theory. Every correct algorithm will behave correctly on cases b) and c) by servicing them and ignoring them respectively. Challenges exist in cases a) and d) where the probability of being in one of the two states is equal and thus the decision depends upon the patterns of service requests.

The metric that we would like to optimise is the behaviour of our reconfigurable machine in respect to the ideal reconfigurable machine, which services the service with maximum speedup for each given time i.e. has zero reconfiguration time.

$$performance = \frac{real\ speedup}{ideal\ speedup}$$

In our simulation we use two services, one of type a) and one of type d) and we observe the performance for different cases. The fact that we have only two doesn't significantly affect the results because this algorithm considers only two services anyway; the best and the currently selected.

Experiment 1: $frame = 20$, $t_{th} = 5ms$, $t_p = 50\,steps$, 10 runs of 10000 time steps each. We run for the combinations of 3 different values of speedups (50 μs, 100 μs, 150 μs) for each process. The results can be seen in the following table and in Figure 27. Statistical errors are within 2% in every case.

		infrequent that last a lot		
	Speedup(μs)	50	100	150
frequent that don't last a lot	50	0.67	0.799	**0.856**
	100	0.781	0.664	0.748
	150	**0.845**	0.729	0.659

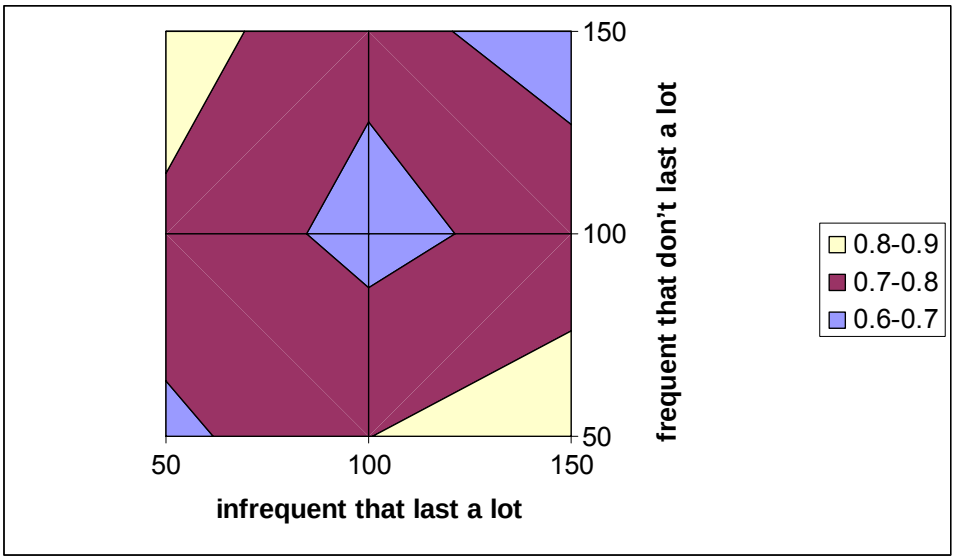


Figure 27. Performance vs speedups

We observe that when the two classes of problems have the same speedup factors, the performance of the reconfiguration manager is the poorest which is what we expect because the two classes are equivalent from the perspective of speedup and thus the error rate is 50%. On the other hand when one class provides more speedup than the other, the performance of the manager gets optimized.

The most important feature is that the performance is symmetrical which means that the reconfiguration manager performs equally well on both classes of services a) and d) with a slight preference on infrequent that last a lot (a) which is reasonable given that they behave more predictably.

Experiment 2: $s_{a)} = 50\mu s$, $s_{d)} = 150\mu s$, $t_{th} = 5ms$, $t_p = 50\,steps$, 10 runs of 10000 time steps each. We run for different $frame$ sizes from 2 to 16 requests. The result can be seen in Figure 28.

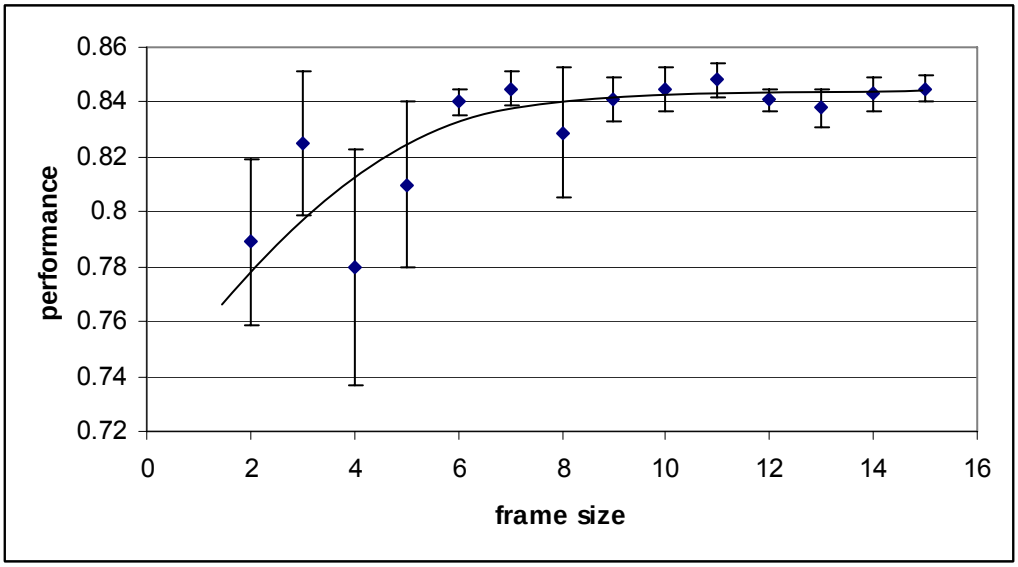


Figure 28. Performance vs frame size

We can see that the frame size doesn't matter as long as it's larger than a certain number which is in our case 9 requests. This is what we expected. For very small size of $frame$ the manager will delay the reconfiguration significantly because the best service S_b changes frequently causing reset of t_{cut_loss}.

Experiment 3: $s_{a)} = 50\mu s$, $s_{d)} = 80\mu s$, $frame = 20$, $t_p = 50\,steps$, 10 runs of 10000 time steps each. We run for different t_{th} from 50 to 7250 μs. The result can be seen in Figure 29.

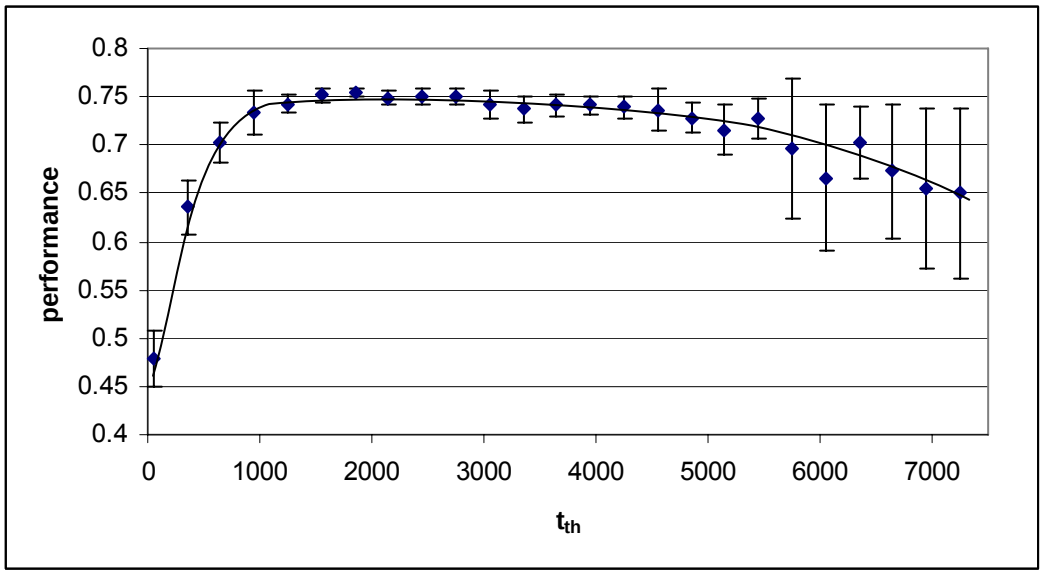


Figure 29. Performance vs t_{th}

We observe again that the value of t_{th} is not important as long as it's larger than 1000 μs and smaller than 5500 μs. If t_{th} is too small, a reconfiguration will be decided too frequently increasing performance losses. If t_{th} is too large, reconfiguration will get delayed resulting performance losses. We can see that the performance for extremely small values of t_{th} is bellow 0.5 which means worst than having a selection at random. This is reasonable considering that with such a low t_{th}, most of the time the service provider will be reconfiguring itself providing no speedup at all. Obviously it's worth having a large value of t_{th}, in the order of reconfiguration time t_p.

Some final comments on the algorithm. The fact that we operate based on the last $frame$ requests acts like a low pass filter that smooths peaks on requests. The fact that the threshold acts on the accumulated t_{loss} (t_{cut_loss}) makes the algorithm decide faster a reconfiguration if it faces huge loss, or slower if the loss is small. For small losses it may not be worth switching and suffering a cost of t_p if you are not guaranteed that the situation is permanent. Finally, although this algorithm is simple it works quite well because it performs on high-quality information that represent directly the structure of the problem and fortunately are available within the service broker module.

3.3 Implementation issues

Here we present implementation details for the most important communication infrastructures at this moment; busses and network-on-chips. Advanced issues like implementation on network-on-chips that support multicasting and connection with off-chip networks are discussed in Appendix E.

3.3.1 On a bus based system (CoreConnect/Amba)

IBM's CoreConnect and ARM's AMBA are both standarized SoC bus interfaces. We can see diagrams taken from their specifications [83-85] in Figure 30 and Figure 31. They both feature a high speed bus (AHB/ASB, PLB) and a low speed buses (APB, OPB/DCR) which get connected through bridges. Arbiters exist in order to manage shared resources.

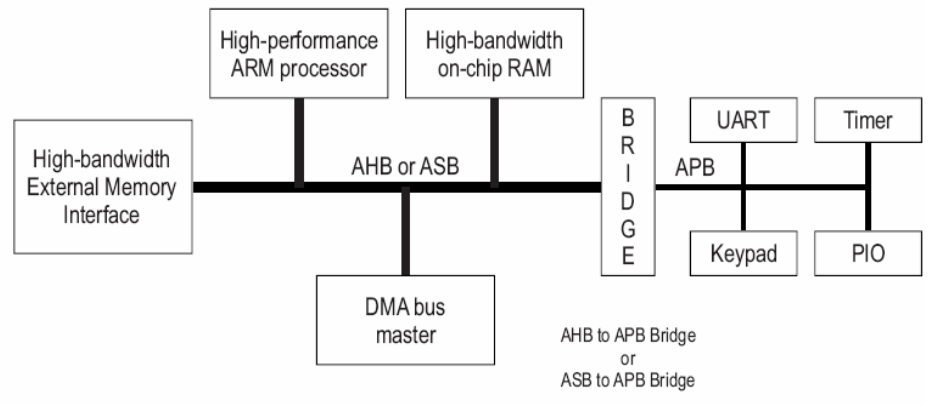


Figure 30. A typical AMBA system

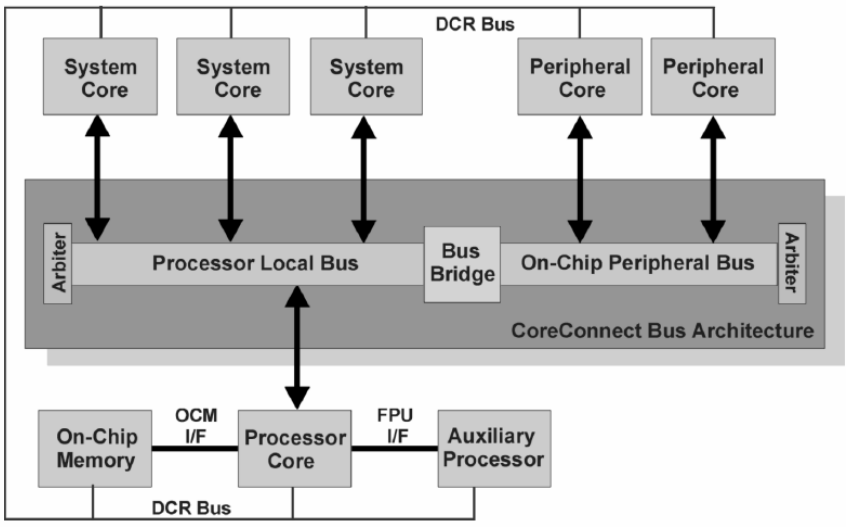


Figure 31. CoreConnect block diagram

Core Services exist for providing acceleration and because communication costs are very important (see 3.4.2) the only place for both the service broker and the service providers is the high speed bus. For compatibility with NoCs the communication between them is going to take place in a serial manner meaning that it uses a small number of memory (or I/O) mapped registers as shown in Figure 32.

	Service broker	Service provider
Data register	SB_BASE_ADDR [R/W]	SPX_BASE_ADDR [R/W]
Status register	SB_BASE_ADDR+1 [R]	SPX_BASE_ADDR+1 [R]

Figure 32. Bus-based architecture registers

The registers are 32-bit long for the processor's programming model even if they are implemented in another way. Figure 33 saws the communications that occur on a bus based system with Core Services.

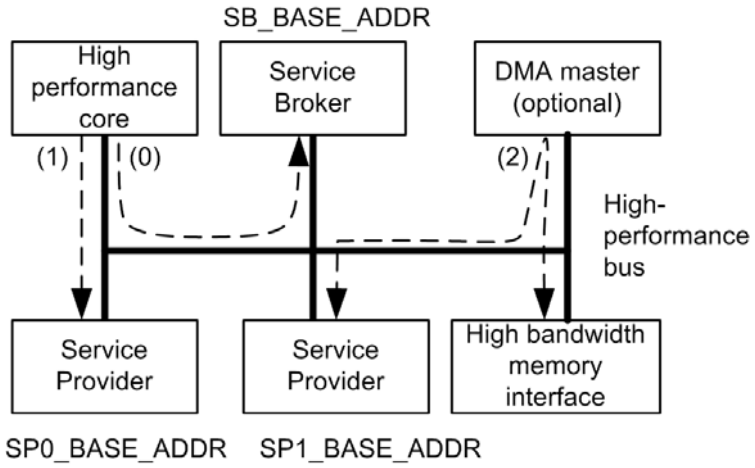


Figure 33. Core Service mechanics on bus-based architecture

Phase I (service request) is initialized by reading the service broker status register (SBSR) in address SB_BASE_ADDR (see Figure 34). This is the only place where inter-processor synchronization is required since the service broker is capable of serving only one processor a time.

MSB		LSB
X	R	B

Figure 34. Service broker status register (SBSR)

If the broker is free, the B(usy) flag is clear and on the read transaction, the flag gets set immediately (atomic operation). Then the requester can start writing word by word the service request sequence (see 3.2.1.1) on the service broker data register (SBDR).

The service requester then polls SBSR and waits until the R(eady) flag is set. Then it can start reading the service assignment response (see 3.2.1.2) on the SBDR word by word. The Provider Unique ID in the response packet is the base address (physical) of the assigned service provider (SPX_BASE_ADDR). After reading the last word, the service broker becomes available and the B(usy) flag gets cleared.

Phase II gets initialized when the service requester sends parameters by writing a request packet (see 3.2.2.1) to each service provider's data register (SPDR) in the address SPX_BASE_ADDR+1. The Requester ID field holds a unique identifier that may be used by the provider to issue an interrupt on the service requester when it completes. If the platform allows multiple slaves to be active at the same time, which

is highly unlikely, this feature could be used to reduce communication costs by broadcasting the parameters.

MSB	LSB
X	D

Figure 35. Service broker status register (SBSR)

By polling the service provider's status register (SPSR) (see Figure 35) the device can know when the computation has completed by checking if the D(one) flag is set. If an interrupt is being used, the interrupt is being cleared by reading SPSR. The response packet (see 3.2.2.2) is being read serially word-by-word by the SPDR. When the last word gets read the D(one) flag of SPSR gets cleared.

When the service completes the providers are registered as free by writing to the SBDR a free resource packet (see 3.2.2.3). The same procedure as with service request (Phase I) must be followed in order to assure atomic operation.

Some final notes. If a DMA master is present in the platform, it can offload the requester from the communication with the provider by using the alternative path (2) instead of (1) shown in Figure 33. Service broker's transactions are small (a few words), the waiting time is small and thus polling is used for communicating with it. On the other hand the large time of computation of service provider makes interrupts more attractive. If there is only one processor the service broker may be implemented within it in software because there is no need of mutli-processor management. Mind that it must be thread-safe.

3.3.2 On networks-on-chip

The two most well known Network-on-Chips are Æthereal [18, 86, 87] from Philips and Xpipes [88]. Their mechanics are similar as described in [10] and their network interfaces (NI) support similar core interfaces (OCP and OCP/DTL/AXI respectivelly). We will assume that there exists a mechanism for sending an array of words (packet) to a processing element unique identified by a number (IP) and a callback mechanism gets invoked when a message with a certain IP is being received. An API with only two functions can implement this mechanism:

noc_write(IP, PACKET_SIZE, PACKET)
noc_callback(PACKET_SIZE, PACKET)

This simple implementation assumes that the Core Services functionality is the only usage of the NoC at least for the nodes involved. Obviously this is not reasonable but it's easy to enrich the two functions with a *port* parameter that allows the NoC to be used for Core Services' purposes only on a specific port number.

By using this API the implementation of Core Services is very easy because they have been designed with NoCs in mind. As we can see in Figure 36 there is a limited amount of IPs that are used by the mechanism.

Service requester	Service broker	Service provider
SR_IP	SB_IP	SPX_IP

Figure 36. IPs used by the Core Services on a NoC architecture

Phase I (service request) is initialized by the service requester sending a service request packet (see 3.2.1.1) to the Service broker (SB_IP). The Requester ID field of the packet must be filled with service requester's IP (SR_IP). There is no need of synchronization as a service broker is receiving and serving only one packet at a given time. Then service broker responds by sending a service assignment response packet (see 3.2.1.2) to the requester by using the SR_IP form the request. The Provider Unique IDs in this packet are the Service Provider's IDs (SPX_IP's).

Then the service requester can initialize Phase II by sending a request packet (see 3.2.2.1) to each SPX_IP. The Requester ID field of the packet must be filled again with service requester's IP (SR_IP). Then each service provider that completes sends a response packet (see 3.2.2.2) back to the service requester by using SR_IP. At the end service requester sends a free resource message (see 3.2.2.3) back to the SB_IP to free the service providers.

3.4 What to make a Core Service?

Obviously no one wants to put engineering effort in making something a Core Service without guaranteeing some performance benefits. Unfortunately the number of factors that contribute to the acceleration is large and guaranteeing it is quite difficult. However there are some theorems and observations that can help us define what is a good candidate for a Core Service.

3.4.1 Estimating speedup margins

Fist of all there is Amdahl's law [89-91] that simply tells us that if we take a function that takes up the 30% of the time and accelerate it 3 times (300% speedup) making it taking 10% of the original total time we have a 25% total system speedup:

$$\frac{70\% + 30\%}{70\% + 10\%} = 1.25$$

Even if we accelerated that function to take no time at all the system acceleration would be 43%. Imagine the engineering effort that could be put on such a huge acceleration on a single function with a moderate system-level result.

You can see the time spent in each function for our demo applications in the following table. From a computational aspect, the AES encryption is more likely to give higher performance benefits from a computational aspect.

Application	Function	Time spent in function
AES encryption	rijndaelEncrypt	65%
Mp3 decoding	synth_full	37%

Amdahl's law in a more formal form states the following:

$$\text{system speedup} = \frac{1}{(1-P) + (P/S)}$$

where P is the proportion of system performance spent on the function that we optimize and S is the speedup that we achieve in that function.

This law allows us to estimate the available improvement margins and help us select the best functions to optimize; the ones that take the most time. The problem is that it

doesn't tell us how much each function can get optimized in advance. This is a very hard problem of algorithm analysis theory but there exist a thesis that can help. It's called parallel computation thesis.

The parallel computation thesis [92] states that if an algorithm uses s(n) storage in a sequential machine, in a parallel machine, it can do the same $s(n)^k$ steps. More formally it states that "the time on any "reasonable" model of parallel computation is polynomially equivalent to sequential space" [93]. Even though evidences of its truth have been given quite early [94], not much progress [95] has been made in recent years for a formal proof.

If we want to paraphrase it a little we can tell that the more extra memory a sequential algorithm uses, the more time it takes on a parallel computer. We are referring to parallel computation because hardware accelerators increase performance by utilizing parallel resources (multipliers, lookup tables etc.). Hardware engineers can intuitively understand that parallel computation thesis is true considering that large memory requirement usually means a lot of state overhead that prevents parallelism.

The worst case of code one can face is the inherently sequential code. This is code that can't get parallelized. Some examples of these kinds of codes are [96]:

- code protected by mutual exclusion in some manner
- conditional critical regions
- monitors
- barriers

In [97] a model is being presented able to classify algorithms as inherently sequential and it's being used to prove that some graph algorithms are inherently sequential. Obviously it's not worth trying to accelerate an algorithm that is inherently sequential.

Practically code segments that can easily get accelerated usually include lookup tables (e.g. state machine implementations, string matching– regular expressions), integer calculations, data intensive mathematical transforms, operations with bit-level manipulations or non-multiple of 8 bit data (e.g. LFSRs) or vector operations. Potential benefits also exist in "FOR loops" with few "IFs" inside.

Warning! Control oriented segments of code (e.g. with a lot of error handling - exceptions, many ifs with variable operands) should be avoided for acceleration.

It should be kept in mind that embedded processors have highly optimized datapaths, efficient caches and usually run on a much higher speed than local buses. In order to anticipate the communication costs, the hardware acceleration has to be massively parallel. Faster interfaces like Xilinx's Auxiliary Processor Unit (APU) controller may lessen in the near future the communication costs and broaden the acceleration margins.

3.4.2 Estimating communication overhead

As presented in [98] the communication costs on a high-performance reconfigurable environment can easily become the performance bottleneck. What they propose as a solution is a pipelining between computation and communication. In [51] they propose to increase the granularity of the hardware accelerated functionality until you

have acceptable levels of communications. This is usually not an option because of the engineering effort it required. Other more advanced techniques like compression could also be worth trying. Although Core Services don't prevent, they also don't enforce the use of any of those techniques. The problem of communication remains and has to be carefully justified to ensure acceleration.

The first problem is that data has to get transferred from the service requester to the service provider. The main argument is that the core would have to read these data anyway, so it's not a pure overhead. The way that an efficient core will send data to the service provider is with a loop like this:

```
1:    LOOP: REG <- [DATA++]
2:    [SERVICE] <- REG
3:    JMP DATA!=SIZE LOOP
```

In most cases core's speed is higher than the bus's speed and thus instruction 2 is the main bottleneck. Instruction 1 may come at free in the (highly likely) case that the data are in cache and would be read by the algorithm anyway. Instruction 3 is negligible especially in the large data transfers we are interested in because the branch is going to be predicted successfully.

A second problem is that the state of the system that is being used by the function must be transferred for each call because Core Services are stateless. Hopefully in most stream operations that are more likely to be chosen for acceleration the amount of state is limited. But it exists. For example, in order to use the AES encryption service we have to send 52 words of key data just to encrypt 4 words of plaintext which if you multiply with the number of calls means that we have to transfer 141Mb of state overhead in order to encrypt an 11Mb long file. We can see in the table bellow statistics on the execution of the two different applications on the same file.

Application	Function	Calls
AES encryption	rijndaelEncrypt	710475
Mp3 decoding	synth_full	14340

In functions with a large number of calls the communication cost and more specifically the state overhead must be careful considered before making them Core Services. Hopefully this overhead is easily predictable for a given platform. Even in the case of Core Services over NoC, the average and worst case latencies can be accurately predicted as shown in [99].

There are solutions for the communication overhead. Using DMA transfers takes away the load of transferring from the process requester. This is useful only if the process requester has something else to do in parallel i.e. runs multiple processes. The drawbacks are that DMA masters take silicon area and lower performance by stealing bus cycles from the processor as presented in [100].

In general the best solution is to map variables that have to be shared with the co-processor to a fast scratch-pad memory. Xilinx OCM-interface for example is traditionally [101, 102] being used for this purpose. By mapping variables there you don't have to move them at all as computations can be done in-place. Both PowerPC and hardware accelerators have single-cycle access in those bi-directional block RAMs. Use of these memories is not supported natively by Core Services. We must note that different processes will have to compete for these scarce resources and explicit compiler and operating system support has to exist for using them [103].

3.4.3 Other aspects

A service provider is allowed to call other Core Services but this feature is highly unlikely to be used in practice because it decreases performance. As a result, a function that has chances of being accelerated must be leaf function i.e. not call any other function.

In some cases, a block in a function needs acceleration while the rest is extremely awkward and inefficient to accelerate e.g. uses a lot of system's space. In these cases, we have to refactor the code slightly and create a function with the block that is efficient to accelerate.

In most of cases readily available software code is hard to parallelize. In [38] Microsoft's chief researchers admit that "leveraging the full power of multicore processors demands new tools and new thinking from the software industry" and they do so in September 2005 when multi-processor cores have already widespread! Writing hardware-friendly software will become a necessity in the next years but until then hardware engineer's job will be hard. Stream processing [104, 105] software is naturally hardware-friendly software and might be an appealing option.

Finally, reconfigurable hardware resources are not infinite. As a result experience and a lot of exploration should be used to solve the trade-off between acceleration and hardware resources, which in some cases will mean lower clock rate and higher power consumption. As demonstrated in [106] with look-up table based decoders found in the JPEG and MPEG protocols the parallel/serial trade-off can be attacked systematically and give efficient results.

Chapter 4. Implementation on a reconfigurable platform

First we will present the implementation of Core Services' framework on this platform (section 4.1) and then demonstrate its usage with the two applications, AES encryption and mp3 decoding (section 4.2).

4.1 Implementation of Core Services on Xilinx's platform

An overview of the Xilinx tools and design flows is being provided in Appendix A. Here we will concentrate on the hardware (section 4.1.1) and software (section 4.1.2) components and the Service Builder (section 4.1.3) used by our Core Services implementation.

4.1.1 Hardware components

A hardware stack has been implemented that abstracts the underlying bus implementations and thus makes the design more portable while at the same time reduces the design time. These designs are fully customizable through a single package (array_types see Appendix B.1) and use extensively generics and generate statements in order to adapt to user requirements. This allows easy customization by the Service Builder platform builder (see section 4.1.3).

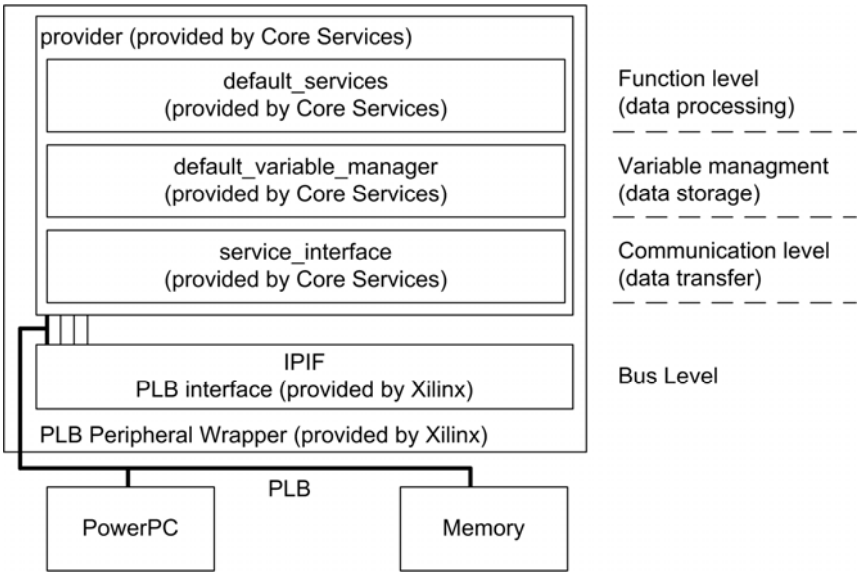


Figure 37. The Core Services' hardware stack over Xilinx's stack

In Figure 37 we see the Core Services' hardware stack over Xilinx's stack. The first level, closer to the PLB bus is the Service Inteface level that handles the communication according to the Core Service's protocol. This is actually a state machine shown in Figure 38 and its VHDL interface can be found in Appendix B.2. This state machine is able to parse the input packets and generate output packets from/to the processor. It creates signals that are suitable for memory like peripherals by providing the Variable Number, equivalent to Chip Enable (CE), the Word equivalent to the Address, the Data and the variable_valid which is equivalent to Write Enable (WE). Some other signals are also used but in many cases they are set to constants. Service Interface handles also protocol-side fault tolerance by featuring the appropriate states into its state machine. It doesn't calculate CRC's itself because

in the case of a bare packet (packet where only CRCs are being sent) computation results are not read at all by the Service Interface and only CRCs are requested from higher levels.

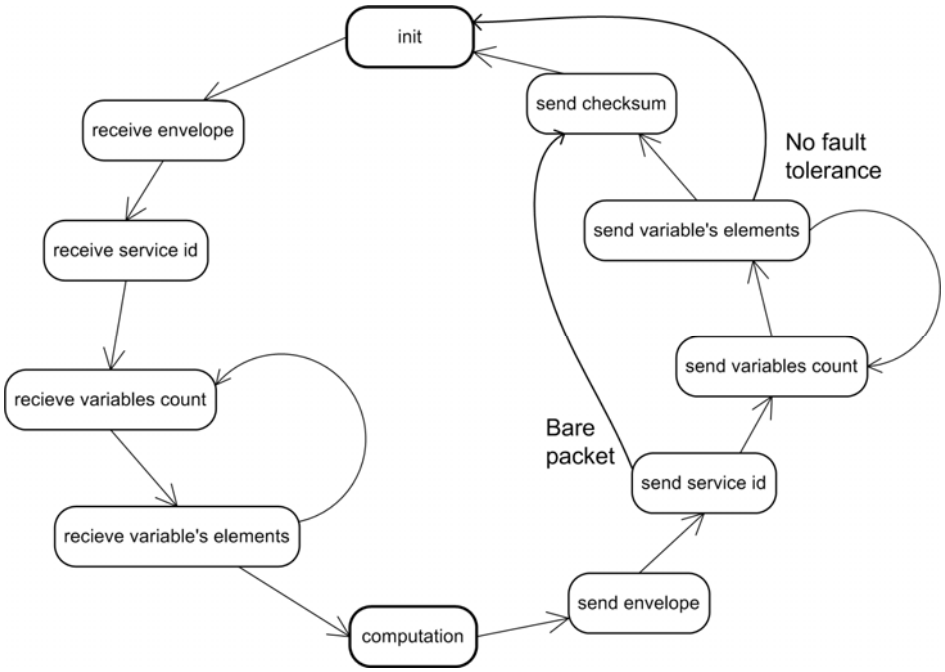


Figure 38. The Service Interface state machine

At the higher level we have the Default Variable Manager component which we can see in Figure 39 and its interface in Appendix B.3.

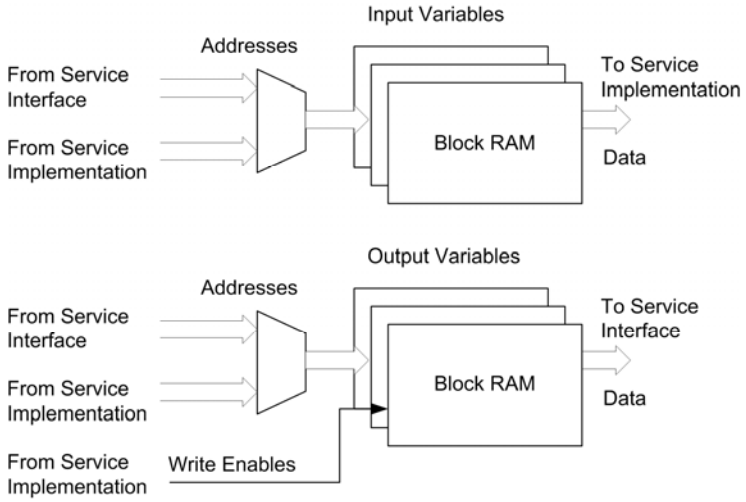


Figure 39. Default Variable Manager

The Default Variable Manager is exactly what the Service Interface would expect to see from component's side. A set of memories, one for each input and output parameter. For each variable at least one Block Ram is being used even if it is one byte long. This is not a problem because plenty of Block Rams are available in every Virtex II FPGA and gives us a significant advantage; a word from each input and output variable can be accessed within a single clock cycle from higher levels. The way that memory and processing speed trade-off has been resolved in our case is

simple but must be suitable for most Service Providers. Of course if something more advanced is required, one can override this level and implement its own variable storage layer (this is what we did for our applications as you can see in the following sections). This component uses heavily GENERATE blocks in order to provide all the customization that is needed regarding the number and size of input/output variables. The interface that it provides to the processing element is that of a master processing element i.e. a processing element that generates addresses for the memories and expects or writes data from/to them.

Exactly that's what Default Services are as we can see in Figure 40 and Appending B.4.

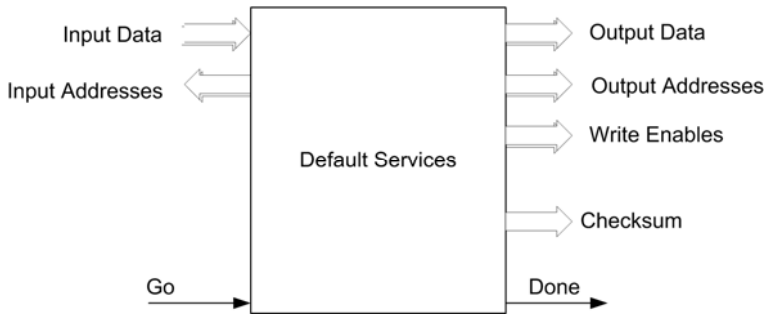


Figure 40. Default Services' Interface

Default Services are provided by the framework as a default implementation that makes vector addition and is useful as a template to rapidly implement new functionalities. Default Services provides a very powerful interface to the processing elements. The power of this interface lies on its simplicity. With Default Services we have the exact equivalent of a software function in hardware. All the input and output variables of the function are available at this level on the hardware and there are no platform dependent signals at all (apart from a single checksum - see Appendix D.1). At this abstraction level the component is really reusable but more importantly this is the hardware abstraction level from which many C to RTL tools start to operate. Obviously this is an important benefit of our framework for realizing accelerated hardware/software co-design flows.

4.1.2 Linux Device Driver and the API

From software side an equivalently large contribution has been made.

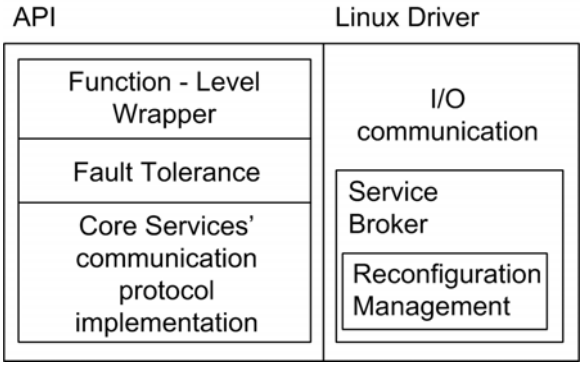


Figure 41. Core Services' software stack

As we can see in Figure 41 and Figure 42 Core Services are implemented with a Linux Driver and an API. There are certain reasons that have influenced these

partitioning. Our platform uses only one PowerPC processor and thus Service Broker need not be external to our device. However, we want our API to be able to work consistently in cases where the Service Broker is external and that's why we have wrapped the service broker within the Linux Driver. This driver provides the API with no more functionality than the one that would be available if the Service Broker was external to the device. The Device driver is also needed to provide I/O operations to our platform. Due to memory protection of Linux, direct I/O operations are not allowed from user space which means that our API can't directly access the bus and the Service Broker or any of the Service Providers (Actually this may be done by using ioperm() but it's very slow and bad practice in general). I/O operations directly affect the performance of the system (see appendix D.2).

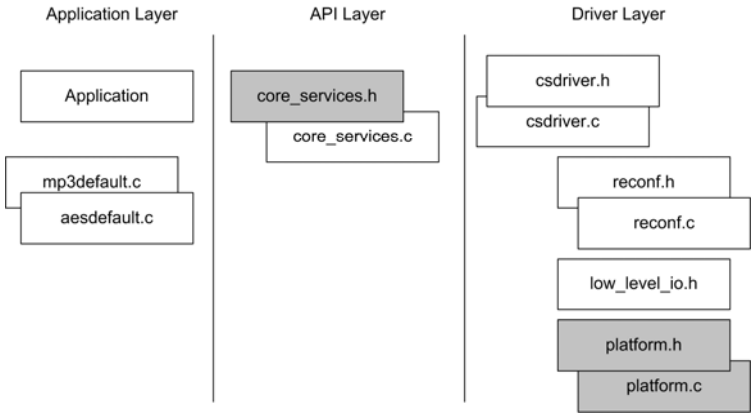


Figure 42. Layers and implementation files. Shaded files are platform specific.

The device driver consists of several files and is written again with a generic way. Platform dependent parameters like the number of Service providers, number and IDs of services, performance costs, input output variables e.t.c. are contained in platform.c and platform.h which allows easy customization from the Service Builder GUI (see section 4.1.3). In the development of the driver precious help was given by the Linux Device Drivers book [107]. It was initially prototyped under SUSE 9.2 (Kernel 2.6) and then ported to MontaVista Linux (Kernel 2.4) with admittedly less trouble than we expected.

```
/dev/csbroker
ioctl() operations:
1. CS_BROKER_REQUEST
2. CS_BROKER_FREE

/dev/cs0 (example: AES accelerator)
/dev/cs1 (example: reconfigurable accelerator)
/dev/cs2 (example: mp3 accelerator)
…

read()/write() operations
```

Figure 43. The interface provided by the Device Driver

The driver provides the interface shown in Figure 43 to the applications. The device /dev/csbroker is the Service Broker and provides the two functionalities discussed in section 3.2.1. Then for each Service Provider, a device is being added in the form /dev/csX where X is an increasing integer number. Which provider corresponds to which hardware component is insignificant since the broker is aware of the sequence and returns adjusted Provider identifiers.

The API provides two levels of functionality. The lower level implements the communication with the Service Broker and the Service Providers. At this level the developer has full control over the actual sequencing of operations like requesting resources and releasing them but must be aware of Core Service's mechanics. An experienced developer can fine tune performance by using the low level API. The high-level API consists of one function declaration for each Service with exactly the same form as described in the Service Builder (see section 4.1.3) and an additional redundancy parameter for Services that require fault tolerance. Obviously the high level API provides a great level of abstraction of the underlying hardware and Core Services' mechanics making development easier and faster.

4.1.3 Service Builder platform generator

As we saw in the previous sections there are various parameters that have to be set on C and VHDL configuration files. Recognizing that customization effort could be a significant disadvantage of the Core Services' methodology we implemented the Service Builder tool in JAVA that one can see in Figure 18 in page 18. With this user-friendly tool the developer can customize all the aspects of Core Services' platform and then all the files are going to be created for him. More specifically one directory is being created for each Service Provider including its VHDL sources customized and ready to be imported to the XPS. Another folder is being created for the software part providing both the Linux device driver and the API customized and ready to get compiled with PowerPC's gcc. This tool significantly simplifies the amount of knowledge needed to apply Core Services and prevents errors thus reducing the development time.

4.2 Applying the methodology on the two demonstration applications

We will now apply this methodology on the two demonstration applications. In each step the process will be described for each of the two applications, AES encoding and mp3 decoding. This way similarities and differences can get highlighted. AES encoding is being done with version 0.7 aescrypt [108] and mp3 decoding is being done with version 0.15.1b of libmad [109]. Testing of the latter is performed with minimad application which comes with the libmad.

Step 1. Profile the system and sort its functions by the total amount of time spent on each.

We performed this step for both applications on a PC by using gprof. Then we verified that the same profile holded true with PowerPC–generated traces examined with the cross-compiled gprof. Both applications were profiled with the same input data (the same mp3 file got decoded and encrypted). The profile for the AES encryption is the following:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
65.09	1.10	1.10	710475	0.00	0.00	rijndaelEncrypt
27.22	1.56	0.46	1388	0.33	0.33	cryptblock
7.69	1.69	0.13				blockEncrypt
0.00	1.69	0.00	1389	0.00	0.00	ewrite
0.00	1.69	0.00	1	0.00	0.00	aes_set_key

The profile for the mp3 decoding is the following:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
32.05	7.30	7.30	14341	0.00	0.00	synth_full
19.53	11.75	4.45	14341	0.00	0.00	output
10.36	14.11	2.36	57364	0.00	0.00	III_huffdecode
9.79	16.34	2.23	1500278	0.00	0.00	III_imdct_l
9.53	18.51	2.17	1032552	0.00	0.00	dct32

We can see that these applications have different difficulties. The first one is slow because rijndaelEncrypt is called too many times and is computationally intensive whereas the second one is slow because synth_full is just computationally intensive. We expect increased communication costs on the first case because of the large number of calls. We also see that the first one offers better potentials of system-level optimizations according to Amdahl's law because 65% of the time is being spent on it.

Step 2. Decide if they are suitable for hardware implementation.

We can see the source code for the two functions on Appendix C.2 and C.3. It is quite difficult to understand how exactly these functions work because they have been identified as performance bottlenecks by software developers and thus optimized (and made cryptic) in several ways. This would make the life really difficult for an automated C to RTL tool. These functions are also almost not documented at all. After some careful examination and several test runs the functionality became clear.

The function rijndaelEncrypt takes an input parameter and xors it with a key for each round. Then it uses each byte of the result to index four different tables T1-T4 which contain the same elements but permutated with a complex way. Then they take the results of those four tables and xor them together producing the output parameter which is used as input parameter for the next round. This operation is being repeted 11 times for 128-bit key. The software developer has optimized the implementation by excluding from the main loop the first and the last rounds that are slightly simpler. As a result we have a complex operation repeated 11 times using constant tables with minor exceptions. In other words; ideal for acceleration.

The function synth_full is not so ideal. First of all it calls another function, the DCT32. That is not a performance bottleneck so there is no need to implement it in hardware. The rest of this function (after DCT32) gets called 2 channels x 36 samples x 14341 calls = 1032552 times and according to profiling it takes on the PowerPC 75μs/call which translates to 8300 clock cycles. By inspecting the source code one can easily see that there is a lot of complexity with many special cases etc. If we calculate the amount of data that has to be transferred we will see that there are: 16xfo tables = 128 words, 1xfx tables = 8 words, 16xfe tables = 128 words, + return 32 words. This means in total 296 words = 5920ns in a 100MHz bus cycle with transfer efficiency of 2cycles/word. Obviously we can save about 15μs with a fast implementation which means 250% function speedup and 25-30% system performance increase according to Amdahl's law.

The problem is that the hardware will have to be complex because we need a state machine able to do all the pointer management. This way the clock cycle might be needed to be lowered and silicon space would be wasted to run sequential code that could run more efficiently on the processor. In order to overcome these drawbacks we apply Core Services in finer granularity. We can see that the code in the function uses actually a single computational primitive two times. The primitive:

```
        ML0(hi, lo, (*fo)[0], ptr[ 0]);
        MLA(hi, lo, (*fo)[1], ptr[14]);
        MLA(hi, lo, (*fo)[2], ptr[12]);
        MLA(hi, lo, (*fo)[3], ptr[10]);
        MLA(hi, lo, (*fo)[4], ptr[ 8]);
        MLA(hi, lo, (*fo)[5], ptr[ 6]);
        MLA(hi, lo, (*fo)[6], ptr[ 4]);
        MLA(hi, lo, (*fo)[7], ptr[ 2]);
        MLN(hi, lo); // Optional
        MLA(hi, lo, (*fe)[0], ptr2[ 0]);
        MLA(hi, lo, (*fe)[1], ptr2[14]);
        MLA(hi, lo, (*fe)[2], ptr2[12]);
        MLA(hi, lo, (*fe)[3], ptr2[10]);
        MLA(hi, lo, (*fe)[4], ptr2[ 8]);
        MLA(hi, lo, (*fe)[5], ptr2[ 6]);
        MLA(hi, lo, (*fe)[6], ptr2[ 4]);
        MLA(hi, lo, (*fe)[7], ptr2[ 2]);
        *pcm1++ = SHIFT(MLZ(hi, lo));
```

or in other words:

$$\sum_{i=0}^{7}\left[\left(fe[i]\cdot ptr1\left[g(i)\right]\right)>>16\right]\pm\left[\left(fo[i]\cdot ptr0\left[f(i)\right]\right)>>16\right]$$

So lets see what we have if we make this primitive a Core Service. We have to transfer 1xfe table = 8 words, 1xfo table = 8 words and return 1 word. This means 17 words = 400 ns with similar conditions as before. This primitive is being called 32 times in each synth_core which means that it takes 75μs/32=2.3 μs. This means that with an efficient implementation we can have a speedup of 160% which translates to a system speedup of 15-20%. Now comes the important part. This implementation uses 296/17=17 times less memory than the previous implementation and also has significantly simpler control logic. We can pessimistically estimate that this implementation is half the size of the previous one. As a result we can fit roughly two such modules in the same area. In a heavy loaded multi-threaded or even better multi-processor environment this means 30-40% increase in system's performance. This module is also significantly easier to create and verify. This is an example of the conclusion drawn in section 3.4.3. There is a trade-off between hardware and acceleration is difficult to identify and optimize.

Step 3. Replace with service calls and provide default service implementation.

This was as easy as replacing the original code segments with function calls with the primitives provided by our Core Services' platform. Some variable renaming was required and some attention in modifying variables in a similar way as the original implementation. For example in case of mp3 decoder the variable pcm1 was increased 16 times inside the loops and by using the function these increases were not reflected back to the original variables (call by value). Thus we had to add manually a pcm1+=16 instruction.

Step 4. Test the software only implementation on the platform.

The software only implementation was tested successfully after fixing minor errors described in the previous step.

Step 5. Calculate the estimated savings of making a hardware accelerator for this function. If constrains aren't yet met, accelerate more functions.

This step aims on industrial applications. In our case we assume that accelerating only the two functions described before satisfies the performance constraints for our system.

Step 6. Create hardware test data.

This step got performed manually by modifying slightly the Core Service's implementation to save function call's data on a file with a VHDL-friendly manner. Future Versions of Core Services may further automate this step by automatically creating VHDL testbenches.

Step 7. Create hardware instance of Core Service using automatically generated service stack and verify with the test data.

The Hardware templates generated by Core Services were used as a starting point for our hardware component device. We decided to customize them by going down to Service Interface level discarding the Default Variable Manager and the Default Service components in order to increase performance and minimize area. By doing so we implemented custom 128-bit wide variables required by the AES algorithm in order to increase performance by accessing more data simultaneously. We also saved memory (area) on the mp3 component because we avoided storing the data on memories completely. Data are being used as soon as they arrive. We also save time with this technique making the component having virtually an impressively small computation time of just 3 clock cycles after the last word arrives!

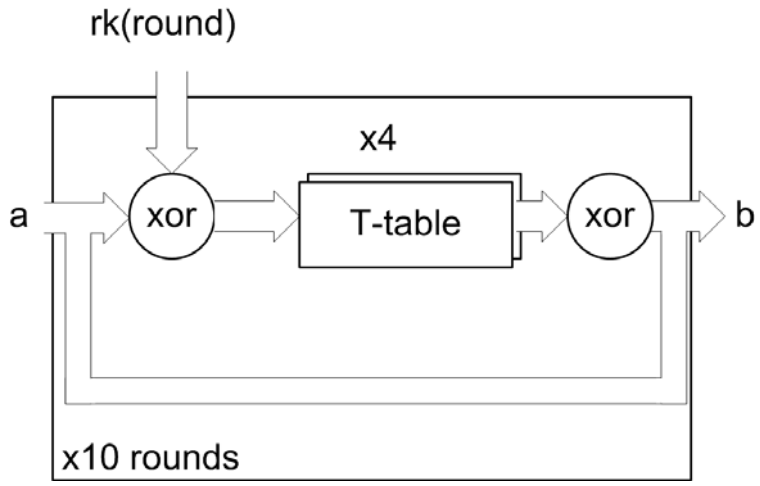


Figure 44. AES accelerator block diagram

The block diagram of the AES component can be seen in Figure 44. Each T-Table is being implemented efficiently by a ROM which translates to a Block Ram on the FPGA. A simple state machine tunes the whole system which is able to complete the block encoding in less than 100 clock cycles. The synthesized core including Core Service's interface uses roughly 3% of the FPGA.

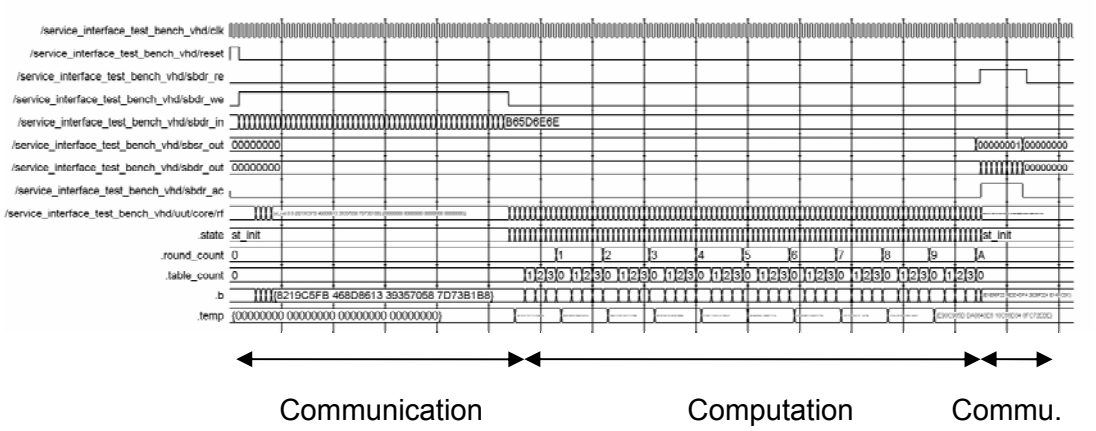


Figure 45. Simulation of the AES accelerator

We can see in Figure 45 a simulation of the AES component. It's impressive that even in this ideal case where one word is being transferred at each clock cycle, communication is almost as time consuming as the computation. By using the bus, about 10 clock cycles are being used for each word transfer which means ten times longer communication.

In Figure 46 we can see the block diagram for the mp3 decoder. This component is more complex than the AES encoder because it is a pipelined implementation. The 32x32 bit hardware multiplier was created by using Xilinx's Core Generator tool and features 5 stages of pipeline internally. In order to access the –synchronous read-constant table that holds the multiplication coefficient (second operant of the multiplication) we need another pipeline stage. Tuning these pipeline stages requires a carefully coded state machine because input data will not arrive in each clock cycle thus input handshaking must be used to stall the pipeline. This implementation completes as we mentioned before in 3 clock cycles and it uses roughly 4% of FPGA's area.

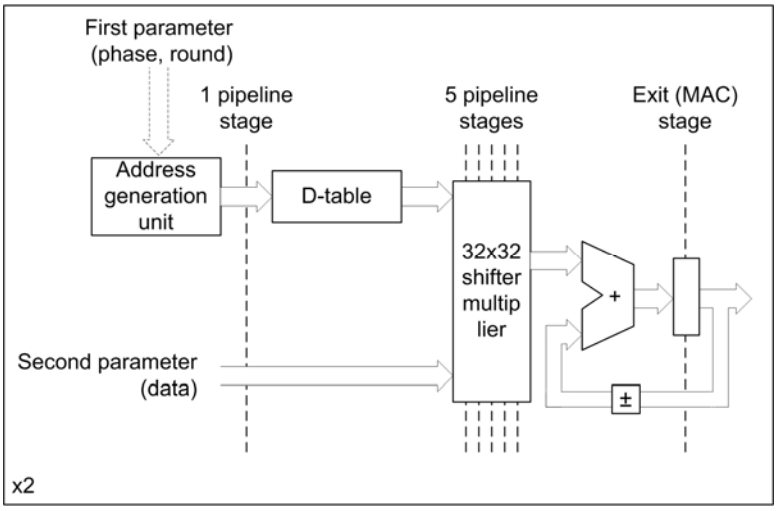


Figure 46. MP3 accelerator block diagram

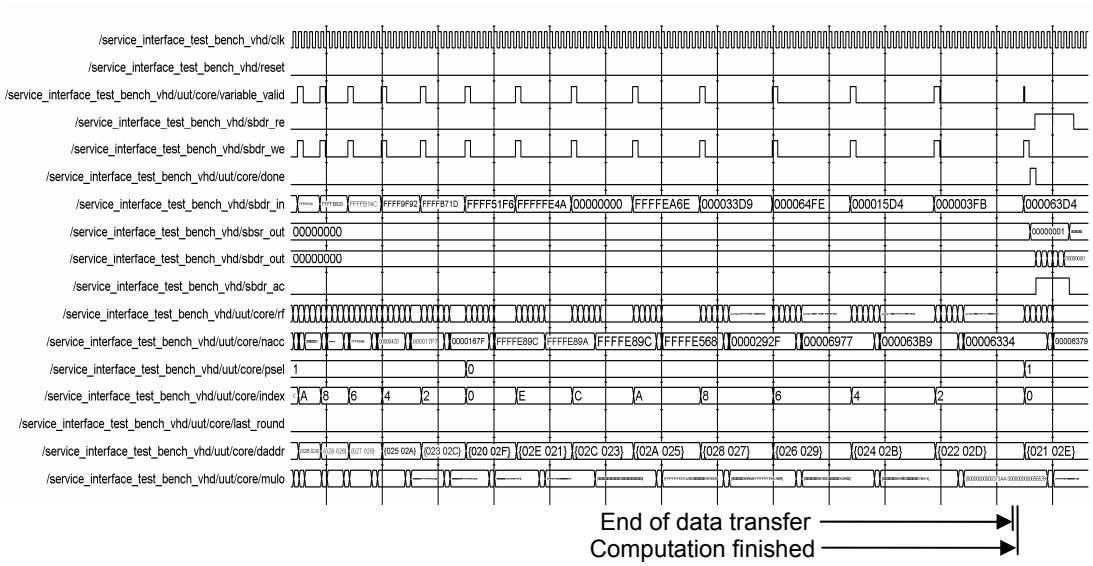


Figure 47. Simulation of the MP3 accelerator

In Figure 47 we can see simulation of the MP3 accelerator. The unequally distributed pulses are the data-ready signals for our design. By providing data with varying rate we verify that the pipeline works correctly. We can see that as soon as the final data arrives the computation completes as expected.

Step 8. Calculate the actual savings of making a function Core Service. If the constrains aren't yet met, create more hardware components or optimize more the existing ones.

As in step 5 this step aims on industrial applications. In this case we assume that performance constraints are met. The benchmarking results are presented in section 5.1.

Step 9. Test the software/hardware implementation on the platform.

Some inconsistencies were found at this level. Because the original test vectors were created under windows on a little endian machine whereas the final implementation was running on a big endian machine (PowerPC) there was a slight incompatibility on the AES module. This of course doesn't mean that C is not portable. The compiler used to issue warnings about incompatible pointer assignments but the original application was not designed with cross-platform compatibility in mind. With slight modifications we converted AES accelerator to big endian and the design completed successfully.

Chapter 5. Evaluation and future work

5.1 Benchmarking and results

We created a system with the following components (see Figure 48); a PowerPC, an AES accelerator, an mp3 accelerator and an (emulated) reconfigurable component able to provide both the mp3 and the AES services.

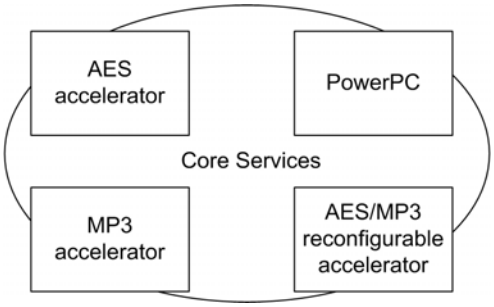


Figure 48. Test system configuration

By using an XPS project we had verified at step #8 of the methodology that hardware implementations, including communication costs were almost three times faster than software implementations (see Appendix D.3). After moving to the Linux environment and by using read/write system calls hardware efficiently was dramatically reduced and software calls became slightly faster than hardware calls. We propose later in this chapter ways to improve this performance. We delayed artificially the software function calls in order to retain the context of “hardware acceleration”. As we mentioned in section 2.4 hardware implementations are not always faster than software if you examine them on a realistic context including communication costs, caches and higher clock frequency of the processor.

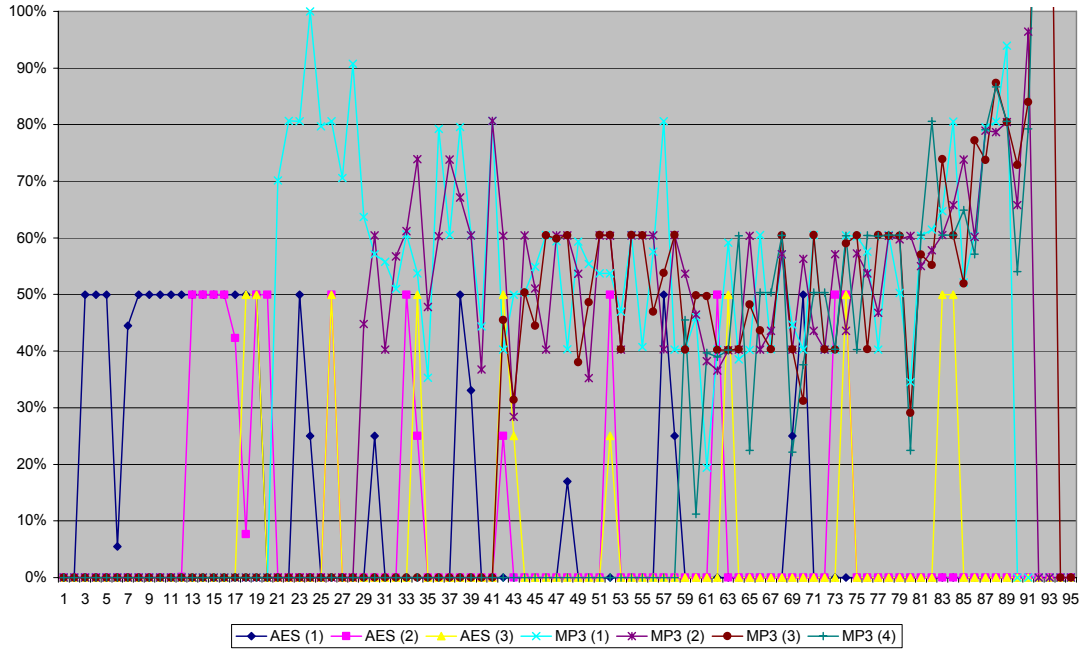


Figure 49. Performance over time with accelerators inactive

In order to evaluate the performance of Core Service's implementations we use a realistic test case. We run three AES encoding and four MP3 decoding instances concurrently. We start one process after another such as transient behaviour can also be studied. We use a utility that we created to measure the throughput. We find the maximum throughputs and then normalize our datasets according to them. The results can be seen for accelerators inactive in Figure 49 and with accelerators active in Figure 50.

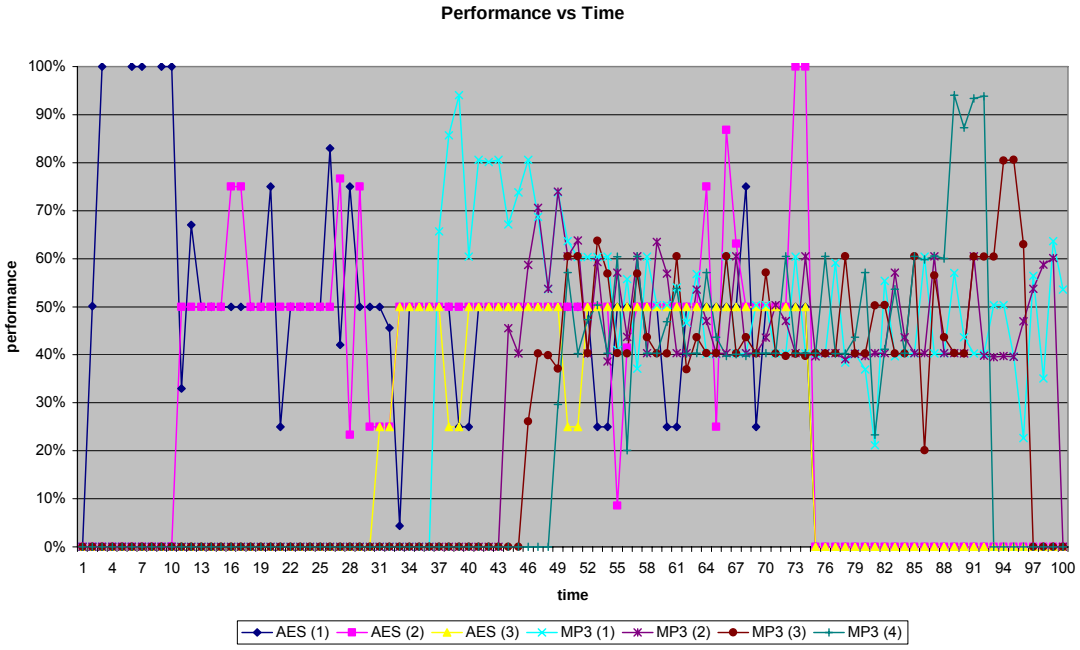


Figure 50. Performance over time with accelerators active

What we actually see is the combined effort of Linux scheduler and Core Service's functionality and that explains the noisy measures. At the beginning we can see the run of the three AES processes. When first AES gets loaded it has the complete focus of the system so it runs on its maximum performance. We can see that in accelerated form it gives 100% of system throughput while in non-accelerated form it gives only 50% of system throughput because software implementation is slower. Then when a second and the third AES process runs in the non-accelerated case Linux's scheduler works on a round-robin fashions and cycles the focus of the processor on each process giving that saw-like profiles (blue, pink, yellow in figure Figure 49). When we have accelerators enabled we can see a completely different profile. When the second and the third AES processes run they all provide the same throughput no matter system's load. This is because hardware acceleration makes computations complete faster and thus scheduler's round robin is smoothed out providing the expected; independent acceleration to all the processes. In the same manner if we had a multi-processor system all the cores would get equally accelerated.

When mp3 processes run we don't get the result we would expect at first-level. We see that in the non-accelerated case we have slightly higher throughput. This is not so unexpected. Linux's scheduler realizes that the AES function is slow and gives priority to the mp3 service. By examining Figure 49 it is clear that mp3 processes "steal" performance from AES encryption. We will show later that despite "stealing" system's performance is superior in the accelerated case. What is also clear is that in the accelerated version there is no visible degrade on AES or MP3 process's

performance no matter how many processes we are running. In the steady state, every process seems to give 50% of its peak performance.

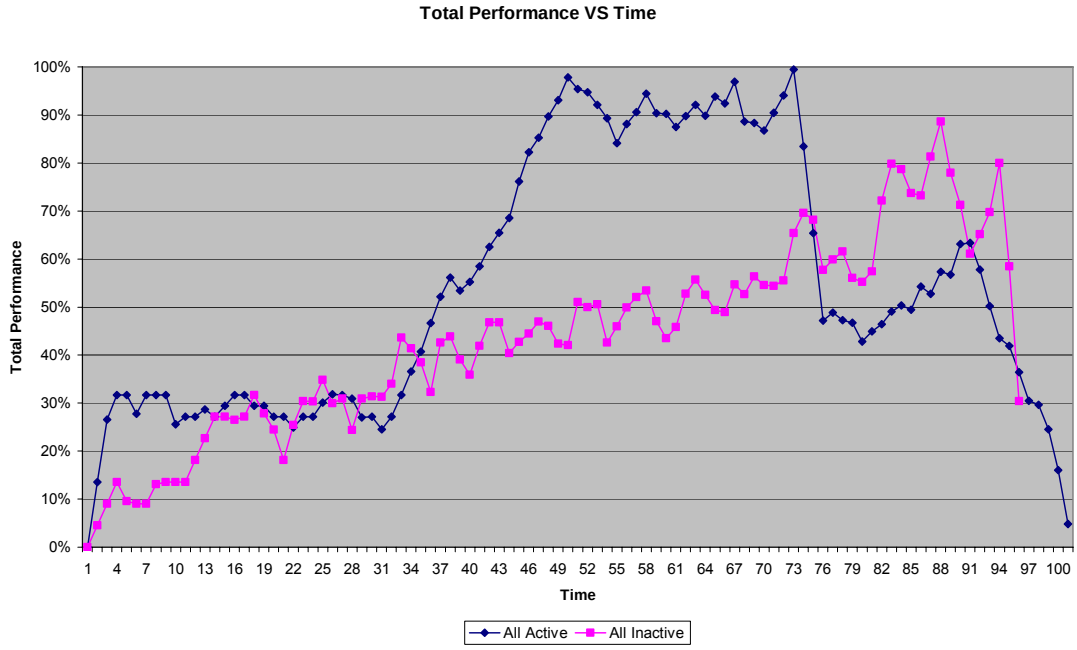


Figure 51. System's performance with active/inactive accelerators

We can see an overview of total performance of the system for these two cases in Figure 51. There is the 3 point moving average of the sum of performances for each case. It's clear that the accelerated version outperforms the non-accelerated especially in the case where we expected it to exceed; a heavily loaded system. Future systems will be heavily loaded not by multiple processes but from multiple processors that will have to compete for the acceleration resources and we can see that Core Services can successfully handle this case.

5.2 Summary

In this dissertation we proposed a new design methodology for multiprocessor system-on-chips inspired by the widely adopted Web Services technology. We specified its mechanics by the means of communication protocols and algorithms and we validated the algorithms with simulation models. Then we described in detail the means in which these mechanisms can be implemented in platforms with communication infrastructures like busses and network-on-chips. Communication protocols were designed with NoCs in mind and thus they have a straightforward efficient implementation on NoCs.

Then we implemented the hardware and software components that are needed for applying this methodology on Xilinx high-end FPGA's platform. More specifically we developed hardware components for communication, data management and function layers that accelerate the design and abstract the underlying bus topology making the component significantly more reusable. We also developed software components including Linux device drivers and two-level API's that make hardware readily available to software designers with minimum effort. The higher level API completely hides the hardware and application's software is actually unaware whether the function is being run on hardware or software. We also developed a JAVA application for automatically generating these software and hardware components customized to our current application's needs as described on our Graphical User Interface.

Obviously the development time for future accelerators on this platform using Core Services will be much shorter by using the Core Services Builder.

Finally the methodology and the platform got verified by applying it to two applications, AES encryption encoding/decoding and mp3 decoding with different levels of granularity. On AES encryption the most time-consuming function was accelerated while in mp3 a sequence of frequently used operations of the most time-consuming function was accelerated. This way we kept the control-intensive operations on the PowerPC and all the data-intensive operations on the hardware accelerators as hardware/software co-design suggests. The two applications were run on a hardware platform consisting of one AES Service Provider, one mp3 Service Provider and one composite Service Provider able to provide both functionalities and treated as reconfigurable component and profiles of throughput on complex test cases were created.

5.3 Conclusion and future work

The key message of this dissertation is clear: Communication costs are of major importance on current and even more future SoCs. The fact that cores and their caches run many times faster than their local busses makes I/O operations significantly more expensive compared to simple CPU operations like arithmetic operations. Hardware acceleration is thus getting increasingly difficult to achieve and can be beneficial only with accelerators with large granularity. These cores impose a large level of control functionality and also have significantly large development cost. Obviously this is not what hardware/software co-design promises. C to RTL compilation tools may shorten design cycles by transforming the high-level behavioural specifications on the form of C functions to hardware descriptions. The first generation of these tools may produce significantly sub-optimal designs in terms of performance and area but the shorter development cycle may make industry adopt these tools rapidly at least for producing initial designs that may be further optimized manually.

The other viable solution for hardware acceleration is on very fine granularity by customizing processors' instruction set. Realizing the problems of coarse grained acceleration Xilinx added another interface on the Virtex 4 and latter devices, the Auxiliary Processor Unit (APU) controller which interfaces directly the CPU pipeline. By using it the designer can extend the instruction set of the PowerPC by creating application specific instructions. The bandwidth provided by the APU controller is much higher than the processor itself transferring up to 16 bytes of data in each instruction. This is a very promising interface and may offer new opportunities for reconfigurability as well because small instruction-level operators feature smaller bitstreams and thus shorter reconfiguration time. The main problem with this interface may be in its software-side integration because most of the compilers assume a fixed instruction set. Interfacing CPU's pipeline may also make debugging difficult and may impose a complex hardware-side interface. It is interesting to explore practically this interface and explore its capabilities and is certainly one of the subjects of our future work.

Another very important subject for future work is the increase on the performance between hardware and software interface. We saw that despite the fact that hardware computation completes rapidly, the communication overhead is the bottleneck that makes hardware implementation less efficient than software implementation. Given that we use the fastest bus available and many optimization techniques have been applied in our source code we should consider current implementation as very good. There are still a few more hardware/software interfaces

that can be evaluated and might give better results. A memory-like interface for the Service Provider might enable the use of the burst mode of the CoreConnect bus between the cache and the component. The implementation may be complex and performance improvements are not guaranteed but it is worth exploring. Another possibility is using the Direct Memory Access (DMA) mechanism and especially on the scattered mode provided on the latest XPS (8.2i). This will release the processor from the load of transferring data to the component and is well supported by Linux (see Chapter 15 in [107]). Using DMA will probably accelerate very large data transfers but it won't provide acceleration on most other cases which are more frequent. Perhaps Xilinx's Core Services implementation should be extended to employ transparently different communication techniques depending on the amount of data transfer required by each Service. Another issue that could be explored is Linux driver to application interface. Now reads/writes on character devices are being used for communication with Service Providers. Network and block driver interfaces could also be explored as well as the ioctl interface that reduces the number of required system calls to one instead of two per service request.

At higher level an implementation of Core Services on a NoC platform would provide interesting insights both on NoCs and Core Services. Core Services define one of the first protocols that aim on NoCs and it is interesting to verify that the approach taken actually fits well current NoC implementations. It is also interesting to see how various NoC characteristics such as connection setup and round-trip time affect the performance of the protocol. By studying communication characteristics like for example frequency of requests and packet sizes of Core Services one can set realistic requirements on the design of a NoC implementation. The traffic that Core Services produce is not an estimation but the actual traffic that will be required from the NoC.

In terms of reconfigurability there are not a lot of issues that need to be verified. When tools will allow easy creation of partial bitstreams and perhaps reconfiguration time becomes shorter (Virtex 4 supports 8 times faster reconfiguration [110]) it will be straightforward to use Core Services with reconfigurable components. The emulation of reconfiguration that we use is equivalent at functional level with true reconfiguration and thus verifying it is interesting just as a proof of concept. In appendix A.2 we give many useful references that will help such an attempt.

Further exploration of the applications of Core Services and usability of the Service Builder platform generator is very important. The limited set of applications that we implemented is sufficient to prove that Core Services work but may not have revealed us the full set of requirements that applications have. The fact that the port of the two applications and the design of the framework were performed by the same person may have made the framework slightly over-designed for those applications. The original codes for the applications came from different sources and both coding styles and computational requirements are quite diverse thus the framework is quite general. Porting more applications will undoubtedly reveal interesting extensions to the Core Services framework.

References

[1] Gordon E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965.

[2] Industry Association, "International Technology Roadmap for Semiconductors: 1999 Edition," presented at International Sematech, Austin, Texas, 1999

[3] Ahmed Amine Jerraya and Wayne Wolf, "Why MPSoCs?," in *Multiprocessors Systems-on-Chips*: Morgan Kaufmann Publishers, 2005, pp. 1-18.

[4] Mary Jane Irwin, Luca Benini, N. Vijaykrishnan, and Mahmut Kandemir, "Techniques for Designing Energy-Aware MPSoCs," in *Multiprocessors Systems-on-Chips*: Morgan Kaufmann Publishers, 2005, pp. 21-48.

[5] Sujit Dey, Kanishka Lahiri, and Anand Raghunathan, "Design of Communication Architectures for High-Performance and Energy Efficient Systems-on-Chips," in *Multiprocessors Systems-on-Chips*: Morgan Kaufmann Publishers, 2005, pp. 187-222.

[6] J. Hu, G. Chen, M. Kandemir, and N. Vijaykrishnan, "Software Power Optimisation," in *System On Chip: Next Generation Electronics*, B. M. Al-Hashimi, Ed., 2006, pp. 289-316.

[7] Richard Goering, "Platform-based design: A choice, not a panacea," in *EE Times*, 2002, [Online]. Available: http://www.eetimes.com/reshaping/platformdesign/OEG20020911S0061.

[8] L. Benini and G. De Micheli, "Powering Networks on Chip," presented at ISSS - International System Synthesis Symposium, 2001, October, pp. 33-38.

[9] L. Benini and G. De Micheli, "Networks on chip: a new paradigm for systems on chip design," presented at Design, Automation and Test in Europe Conference and Exhibition, 2002, March pp. 418-419.

[10] Luca Benini and Davide Bertozzi, "Network-on-chip architectures and design methods," in *System On Chip: Next Generation Electronics*, B. M. Al-Hashimi, Ed., 2006, pp. 589-624.

[11] J. Nurmi, "Network-on-Chip: A New Paradigm for System-on-Chip Design," presented at International Symposium on System-on-Chip, International Symposium on System-on-Chip, 2005, pp. 2-6.

[12] K. Virk and J. Madsen, "A system-level multiprocessor system-on-chip modelling framework.," presented at International Symposium on System-on-Chip (ISSoC), Tampere, Finland, 2004, Nov.

[13] Luciano Bononi and Nicola Concer, "Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh," presented at Design, Automation, and Test in Europe 2006

[14] Doris Ching, Patrick Schaumont, and Ingrid Verbauwhede, "Integrated Modeling and Generation of a Reconfigurable Network-on-Chip," presented at 18th International Parallel and Distributed Processing Symposium (IPDPS'04), 2004, pp. 139b.

[15] M.P. Vestias and H.C. Neto, "Co-synthesis of a configurable SoC platform based on a network on chip architecture," presented at Asia and South Pacific Conference on Design Automation, 2006, Jan.

[16] T. Hollstein, H. Zimmer, and M. Glesner, "Dynamic hardware/software co-design based on a communication-centric hyper-platform," presented at The 16th International Conference on Microelectronics, 2004, Dec., pp. 355-358.

[17] OpenCores organization, "WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision: B.3," pp. 98, 2002, September 7.

[18] Andrei Radulescu, John Dielissen, Santiago Gonzalez Pestana, Om Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens, "An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming," presented at IEEE Transactions on CAD of Integrated Circuits and Systems, 2005, January, vol. 24(1)

[19] Jawad Khan and Ranga Vemuri, "Battery-Efficient Task Execution on Reconfigurable Computing Platforms with Multiple Processing Units," presented at 19th IEEE International Parallel and Distributed Processing Symposium 2005, vol. 4, pp. 155.

[20] S. Evain and J.-P. Diguet, "From NoC security analysis to design solutions," presented at IEEE Workshop on Signal Processing Systems Design and Implementation, 2005, Nov., pp. 166-171.

[21] Michalis D. Galanis, Athanasios Milidonis, George Theodoridis, Dimitrios Soudris, and Constantinos E. Goutis, "A Framework for Partitioning Computational Intensive Applications in Hybrid Reconfigurable Platforms," presented at International Parallel and Distributed Processing Symposium 2005

[22] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis, "Performance improvements using coarse grain reconfigurable logic in embedded SoCs," presented at Field-Programmable Custom Computing Machines CA, USA, 2005, pp. 630-635.

[23] M.D. Galanis, A. Milidonis, G. Theodoridis, D. Soudris, and C.E. Goutis, "A Framework for Partitioning Computational Intensive Applications in Hybrid Reconfigurable Platforms," presented at IPDPS, 2005

[24] P. Schaumont, K. Sakiyama, A. Hodjat, and I. Verbauwhede, "Embedded software integration for coarse-grain reconfigurable systems," presented at 18th International Parallel and Distributed Processing Symposium, 2004, April, pp. 137.

[25] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung, "Reconfigurable computing: architectures and design methods," in *System On Chip: Next Generation Electronics*, B. M. Al-Hashimi, Ed., 2006, pp. 452-493.

[26] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," presented at DATE, 2001, pp. 642-649.

[27] R. Hartenstein, "Trends in Reconfigurable Logic and Reconfigurable Computing," presented at 9th Int'l Conf. Electronics Circuits Systems, 2002, vol. 2

[28] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," presented at Computers and Digital Techniques, 2006, May, vol. 153(3), pp. 157-164.

[29] Yan-Xiang Deng, Chao-Jang Hwang, and Der-Chyuan Lou, "Two-Stage Reconfigurable Computing System Architecture," presented at 18th International Conference on Systems Engineering, 2005, pp. 389-394.

[30] Wenyin Fu and Katherine Compton, "An Execution Environment for Reconfigurable Computing," presented at EEE Symposium on Field-Programmable Custom Computing Machines, 2005

[31] M. Majer, C. Bobda, A. Ahmadinia, and J. Teich, "Packet Routing in Dynamically Changing Networks on Chip," presented at 19th IEEE International Parallel and Distributed Processing Symposium, 2005, April pp. 154b - 154b.

[32] A. Ahmadinia, C. Bobda, J. Ding, M. Majer, J. Teich, S.P. Fekete, and J.C. van der Veen, "A practical approach for circuit routing on dynamic reconfigurable devices," presented at The 16th IEEE International Workshop on Rapid System Prototyping, 2005, June., pp. 84-90.

[33] Christophe Bobda, Ali Ahmadinia, Mateusz Majer, Jürgen Teich, Sándor P. Fekete, and Jan van der Veen, "DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices," presented at FPL, 2005, pp. 153-158.

[34] Christophe Bobda and Ali Ahmadinia, "Dynamic Interconnection of Reconfigurable Modules on Reconfigurable Devices," *IEEE Design & Test*, vol. 22(5), 2005

[35] R. Soares, I.S. Silva, and A. Azevedo, "When reconfigurable architecture meets network-on-chip," presented at 17th Symposium on Integrated Circuits and Systems Design, 2004, Sept., pp. 216-221.

[36] R. Hecht, S. Kubisch, A. Herrholtz, and D. Timmermann, "Dynamic reconfiguration with hardwired networks-on-chip on future FPGAs," presented at International Conference on Field Programmable Logic and Applications, 2005, Aug., pp. 527-530.

[37] A. Kumar, I. Ovadia, J. Huiskens, H. Corporaal, and J. van Meerbergen, "Reconfigurable Multi-Processor Network-on-CHip on FPGA," presented at ASCI, 2006

[38] Herb Sutter and James Larus, "Software and the Concurrency Revolution," *ACM Queue*, vol. 3, no. 7, 2005, September

[39] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie, "Unbounded Transactional Memory," presented at 11th International Symposium on High-Performance Computer Architecture, 2005, February pp. 316-327.

[40] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," presented at 32nd Annual International Symposium on Computer Architecture, 2005

[41] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy, "Energy reduction in multiprocessor systems using transactional memory," presented at International Symposium on Low Power Electronics and Design, 2005, pp. 331-334.

[42] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," presented at 31st Annual International Symposium on Computer Architecture, 2004

[43] Janice M. Stone, Harold S. Stone, Phil Heidelberger, and John Turek, "Multiple Reservations and the Oklahoma Update," presented at IEEE Parallel & Distributed Technology, 1993, November pp. 58-71.

[44] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," presented at ISCA, 1993, May

[45] S. Stuijk, T. Basten, B. Mesman, and M. Geilen, "Predictable embedding of large data structures in multiprocessor networks-on-chip," presented at 8th Euromicro Conference on Digital System Design, 2005, Sept., pp. 388-395.

[46] World Wide Web Consortium (W3C), "Web Services Architecture," 2004, February [Online]. Available: http://www.w3.org/TR/ws-arch.

[47] World Wide Web Consortium (W3C), "Simple Object Access Protocol (SOAP) 1.1," 2003, [Online]. Available: http://www.w3.org/TR/soap.

[48] Inc UserLand Software, "XML-RPC Specification," 2003, [Online]. Available: http://www.xmlrpc.com/spec.

[49] World Wide Web Consortium (W3C), "Web Services Description Language (WSDL) 1.1," 2001, [Online]. Available: http://www.w3.org/TR/wsdl.

[50] OASIS consortium, "OASIS UDDI Specifications TC," 2002, [Online]. Available: http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.

[51] Ivan Gonzalez, Javier Sanchez-Pastor, Jorge L. Hernandez-Ardieta, Francisco J. Gomez-Arribas, and Javier Martínez, "Using Reconfigurable Hardware Through Web Services in Distributed Applications," presented at

Field Programmable Logic and Applications: 14th International Conference, 2004, August, pp. 1110-1112.

[52] William J. Dally and Brian Towles, "Route packets, not wires: on-chip interconnection networks," presented at Design Automation Conference (DAC), Las Vegas, NV, 2001, June pp. 684-689.

[53] Bart Vermeulen, John Dielissen, Kees Goosens, and Kalin Ciordas, "Bringing Communication Networks on chips: Test and Verification Implications," *IEEE Communications Magazine*, vol. 41(9), pp. 74-81, 2003, September

[54] A. Bartic, D. Desmet, J. Mignolet, J. Miller, and F. Robert, "Mapping concurrent applications on Network-on-Chip platforms," presented at IEEE Workshop on Signal Processing Systems - SIPS, Athens, Greece, 2005, pp. 154-159.

[55] K. Srinivasan, K. S. Chatha, and G. Konjevod, "Linear programming based techniques for synthesis of network-on-chip architectures," presented at Int. Conf. Comput. Des.,, 2004, pp. 422-429.

[56] G.Ascia, V.Catania, and M.Palesi, "An Evolutionary Approach to Network-on-Chip Mapping Problem," presented at IEEE Congress on Evolutionary Computation, Edinburgh, UK, 2005, September

[57] Hu Jingcao and R. Marculescu, "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints," presented at Design, Automation and Test in Europe Conference and Exhibition, 2004, Feb, vol. 1, pp. 234-239.

[58] J. Hu and R. Marculescu, "Communication and Task Scheduling of Application-Specific Networks-on-Chip," presented at Computers & Digital Techniques, 2005, Sep.

[59] Wu Chia-Ming, Chi Hsin-Chou, and Lee Ming-Chao, "Mapping of IP cores to network-on-chip architectures based on communication task graphs," presented at 6th International Conference On ASIC, 2005, Oct., pp. 953- 956.

[60] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "Mapping and configuration methods for multi-use-case networks on chips," presented at Asia and South Pacific Conference on Design Automation, 2006, Jan

[61] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "A Methodology for Mapping Multiple Use-Cases onto Networks on Chips," presented at Design, Automation and Test in Europe DATE '06, 2006, March

[62] Peng Yang, Paul Marchal, Chun Wong, Stefaan Himpe, Francky Catthoor, Patric David, Johan Vounckx, and Rudy Lauwereins, "Cost-Efficient Mapping of Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems," in *Multiprocessors Systems-on-Chips*: Morgan Kaufmann Publishers, 2005, pp. 313-335.

[63] B. Ahmad, A.T. Erdogan, and S. Khawam, "Architecture of a Dynamically Reconfigurable NoC for Adaptive Reconfigurable MPSoC," presented at First NASA/ESA Conference on Adaptive Hardware and Systems, 2006, June pp. 405-411.

[64] M. Ali, M. Welzl, M. Zwicknagl, and S. Hellebrand, "Considerations for fault-tolerant network on chips," presented at The 17th International Conference on Microelectronics, 2005

[65] M. Pirretti, G.M. Link, R.R. Brooks, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin, "Fault tolerant algorithms for network-on-chip interconnect," presented at IEEE Computer society Annual Symposium on VLSI, 2004, Feb., pp. 46-51.

[66] M. Ali, M. Welzl, and S. Hellebrand, "A dynamic routing mechanism for network on chip," presented at 23rd NORCHIP Conference, 2005, Nov.

[67] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an Operating System for a Heterogeneous Reconfigurable SoC," presented at RAW'03 workshop, 2003

[68] Kiran Puttegowda, David I. Lehn Bradley, Jae H. Park, Peter Athanas, and Mark Jones, "Context Switching in a Run-Time Reconfigurable System," *The Journal of Supercomputing*, pp. 239 - 257, 2003.

[69] L. Bubb, C. Pimlott, K. Rees, M. Stewart, and J. Yates, "A Run-Time Support Environment for Reconfigurable Systems," presented at Euromicro Symposium on Digital Systems Design, 2001, pp. 135.

[70] B. Randell, P. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *ACM Computing Surveys*, pp. 123 - 165, 1978.

[71] Alireza Ejlali, Bashir M. Al-Hashimi, Marcus T. Schmitz, Paul Rosinger, and Seyed Ghassem Miremadi, "Combined Time and Information Redundancy for SEU-Tolerance in Energy-Efficient Real-Time Systems," presented at IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2006, April, pp. 323 - 335.

[72] Carl Carmichael, "Correcting Sinle Event Upsets Through Virtex Partial Configuration," 2000, June.

[73] Alberto Sangiovanni-Vincentelli, "Defining platform-based design," in *EE Times*, 2002, [Online]. Available: http://www.eetimes.com/news/design/showArticle.jhtml?articleID=16504380.

[74] M. Goudarzi, S. Hessabi, and A. Mycroft, "Overhead-free polymorphism in network-on-chip implementation of object-oriented models," presented at Design, Automation and Test in Europe Conference and Exhibition, 2004, Feb., vol. 2, pp. 1380-1381.

[75] M. Goudarzi, S. Hessabi, and A. Mycroft, "Object-oriented ASIP Design and Synthesis," presented at Forum on Specification and Design Languages, Frankfurt., 2003, Sept.

[76] Giovanni Agosta, Francesco Bruschi, Marco Santambrogio, and Donatella Sciuto, "A Data Oriented Approach to the Design of reconfigurable Stream Decoders," presented at 3rd Workshop on Embedded Systems for Real-Time Multimedia, 2005, Sept., pp. 107- 112.

[77] Michael Ullmann and Jórgen Becker, "Communication Concept for Adaptive Intelligent Run-Time Systems Supporting Distributed Reconfigurable Embedded Systems," presented at 20th International Parallel and Distributed Processing Symposium - IPDPS 2006. , 2006, April.

[78] Ronald Hecht, Stephan Kubisch, Harald Michelsen, Elmar Zeeb, and Dirk Timmermann, "A Distributed Object System Approach for Dynamic Reconfiguration," presented at 20th Parallel and Distributed Processing Symposium - IPDPS 2006. , 2006, April., pp. 8.

[79] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J.-Y. Mignolet, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," presented at Design, Automation and Test in Europe, 2005, pp. 234-239.

[80] J.T. Russell and M.F. Jacome, "Software power estimation and optimization for high performance, 32-bit embedded processors," presented at International Conference in Computer Design, 1998, Oct., pp. 328-333.

[81] Amit Sinha and Anantha P. Chandrakasan, "JouleTrack A Web Based Tool for Software Energy Profiling," presented at Design Automation Conference, 2001

[82] Amit Sinha, Nathan Ickes, and Anantha P. Chandrakasan, "Instruction Level and Operating System Profiling for Energy Exposed Software," presented at IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2003, December., pp. 1044-1057.

[83] ARM, "AMBA Specification (Rev 2.0)," 1999.

[84] ARM, "AMBA AXI Protocol v1.0," 2004.
[85] IBM, "CoreConnect Bus Architecture," 2001.
[86] Kees Goossens, John Dielissen, and Andrei Radulescu, "The Aethereal network on chip: Concepts, architectures, and implementations," presented at IEEE Design and Test of Computers, 2005, Sept-Oct, vol. 22(5), pp. 21-31.
[87] Chris Bartels, Jos Huisken, Kees Goossens, Patrick Groeneveld, and Jef van Meerbergen, "Comparison of An Aethereal Network on Chip and A Traditional Interconnect for A Multi-Processor DVB-T System on Chip," presented at Proc. IFIP Int'l Conference on Very Large Scale Integration (VLSI-SoC), 2006, October.
[88] Davide Bertozzi and Luca Benini, "Xpipes: A Network-on-Chip architecture for Gigascale Systems-on-Chip," *IEEE circuits and systems magazine*, pp. 18-31, 2004.
[89] G.M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," presented at AFIPS Conference, 1967, Apr., vol. 30, pp. 483-485.
[90] John L. Gustafson, "Reevaluating Amdahl's Law," in *CACM*, vol. 31(5), 1988, pp. 532-533, [Online]. Available: http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html.
[91] Yuan Shi, "Reevaluating Amdahl's Law and Gustafson's Law," 1996, October., [Online]. Available: http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html.
[92] A.K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation," *ACM Queue*, vol. 28, no. 1, pp. 114-133, 1981, Jan.
[93] Ian Parberry, "Parallel speedup of sequential machines: A defence of the parallel computation thesis," *SIGACT News*, 1986.
[94] Leslie M. Goldschlager, "A universal interconnection pattern for parallel computers," *ACM Queue*, vol. 29(4), pp. 1073-1086, 1982, October.
[95] Ryan Williams, "Parallelizing time with polynomial circuits," presented at 17th ACM symposium on Parallelism in algorithms and architectures, 2005, pp. 171-175.
[96] Arnold N. Pears, "CS3 parallel Computing: Lecture 15," *Uppsala University*, 1996.
[97] Raymond Greenlaw, "A Model Classifying Algorithms as Inherently Sequential with Applications to Graph Searching," *Information and Computation* vol. 97, pp. 133-149, 1992.
[98] E. El-Araby, M. Taher, K. Gaj, T. El-Ghazawi, D. Caliga, and N. Alexandridis, "System-level parallelism and throughput optimization in designing reconfigurable computing applications," presented at 18th International Parallel and Distributed Processing Symposium, 2004, April., pp. 136.
[99] A. Jantsch, "Models of Computation for Networks on Chip," presented at Sixth International Conference on Application of Concurrency to System Design - ACSD 2006, 2006, June.
[100] Tai-Yi Huang, Jane W.-S. Liu, and David Hull, "A method for bounding the effect of DMA I/O interference on program execution time," presented at Real-Time Systems Symposium, 1996, December . pp. 275-285.
[101] Ivo Bolsens, "Challenges and opportunities for FPGAs," 2003, July.
[102] Xilinx, "Xilinx Solutions for Network Test & Measurement," 2005.
[103] Ahmed Amine Jerraya and Wayne Wolf, "Memory Systems and Compiler Support for MPSoC Architectures," in *Multiprocessors Systems-on-Chips*: Morgan Kaufmann Publishers, 2005, pp. 251-282.
[104] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz, "The Stream Virtual Machine," presented at 13th International Conference on Parallel Architectures and Compilation Techniques, 2004, pp. 267-277.

[105] Jayanth Gummaraju and Mendel Rosenblum, "Stream Programming on General-Purpose Processors," presented at 38th annual IEEE/ACM International Symposium on Microarchitecture, Barcelona, Spain, 2005, pp. 343-354.
[106] Claus Schneider, "A Parallel/Serial Trade-Off Methodology for Look-Up Table Based Decoders," presented at Design Automation Conference, 1997, pp. 498-503.
[107] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers*, 3rd ed, 2005, February.
[108] Eric Lee Green and Randy Kaelber, "AESCrypt: Rijndael encryption for shell scripts and Ruby.," [Online]. Available: http://aescrypt.sourceforge.net/.
[109] Inc. Underbit Technologies, "MAD: MPEG Audio Decoder," [Online]. Available: http://www.underbit.com/products/mad/.
[110] Adam Donlin, "New tools for FPGA Dynamic Reconfiguration," in *Xilinx Research*, 2005, December.
[111] Xilinx, "DS083: Virtex-II Pro and Virtex-II Pro X Platform FPGAs:Complete Data Sheet," 2005, Octomber.
[112] Xilinx, "UG069: Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual," 2005, March.
[113] Xilinx, "Embedded System Tools Reference Manual," 2005, July.
[114] Xilinx, "OS and Libraries Document Collection ", 2005, July.
[115] Andy Norton, "Using Xilinx Embedded Processor Subsystems in a Synplify Design Flow," 2005, May.
[116] Xilinx, "Platfrom Studio User Guide," 2005, February.
[117] Xilinx, "XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based," 2004, September.
[118] Xilinx, "Development System Reference Guide," pp. 81-107, 2001.
[119] Gregory Mermoud, "A module-Based Dynamic Partial Reconfiguration tutorial," 2004, November.
[120] Geert Braeckeman, Gerd Van den Branden, Abdellah Touhafi, and Geoffroy Van Dessel, "Module Based Partial Reconfiguratio: a quick tutorial," 2004, July.
[121] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," presented at FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 2006, May
[122] Abdellah Touhafi, Geert Braeckeman, and Van Dessel Geoffroy, "Module Based Partial and Dynamic Reconfiguration," 2006, March.
[123] Mark Ng and Mike Peattie, "XAPP502: Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMap Mode," 2002, November.
[124] Xilinx, "Xilinx Device Drivers Documentation," 2004, Jun.
[125] Jamie Lin, "Porting Linux to XUPV2P," 2006, February, [Online]. Available: http://www.eecs.wsu.edu/~jamie/research/LinuxPort/linux_port.htm.
[126] Jamie Lin, "Linux GPIO Device Drivers," 2006, Janurary, [Online]. Available: http://www.eecs.wsu.edu/~jamie/research/LinuxPort/gpio_driver.htm.
[127] Lu Zhonghai, Yin Bei, and A. Jantsch, "Connection-oriented multicasting in wormhole-switched networks on chip," presented at IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures, 2006, March.
[128] Nir Arad, "Philips Æthereal Network on Chip," presented at VLSI Architecture Seminar, 2006, pp. 57.
[129] Wesley Chou, Vicki Shimizu, and Rolf Muralt, "Tiny Tera High Performance Switching," [Online]. Available: http://tiny-tera.stanford.edu/tiny-tera.

Appendix A. Overview of Xilinx's hardware, tools and design flows

A.1 Hardware and tools overview

Xilinx provides us with a set of tools and flows to work with its reconfigurable hardware.

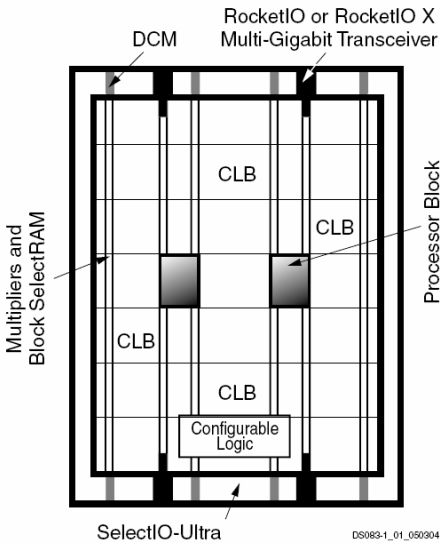


Figure 52. Architecture overview of Virtex II Pro FPGA

The family of FPGAs that we work with is the Xilinx Virtex II Pro FPGAs (see Figure 52 taken from [111]) that include plenty of resources, including (in XC2VP30) two PowerPC hard cores, 20000 reconfigurable slices, 140 embedded multipliers and Block RAMs, 8 Digital Clock Management Units and more than 600 I/Os. Large designs can fit in these FPGAs and hardware software co-design techniques can be applied because PowerPC's can run software and reconfigurable logic can implement hardware functions.

We use the Xilinx University Program Virtex-II Pro Development System that provides us a lot of useful external peripherals (see Figure 53 taken from [112]). The ones that we use are the Compact Flash controller in order to load configuration and boot Linux from a Compact Flash memory, the RS232 ports to communicate with a host computer and the DDRAM controller that provides us 256 Mb of external memory. OF course, we implicitly use the 100MHz System Clock and the JTAG interface for configuration.

These powerful hardware resources need flexible software in order to reveal their strengths and this is the case with Xilinx's software tools. The ISE development environment (see Figure 54) is ideal for small designs and component creation and validation because it can easily run Xilinx's traditional design flow that includes synthesis, translation, mapping, place and route and load into the FPGA. For platform-level applications Xilinx provides the advanced development environment Xilinx Platform Studio (XPS) [113] as we can see in Figure 55. With this complete systems can be created using PowerPC/Microblaze processors, Xilinx's IP cores and custom peripherals.

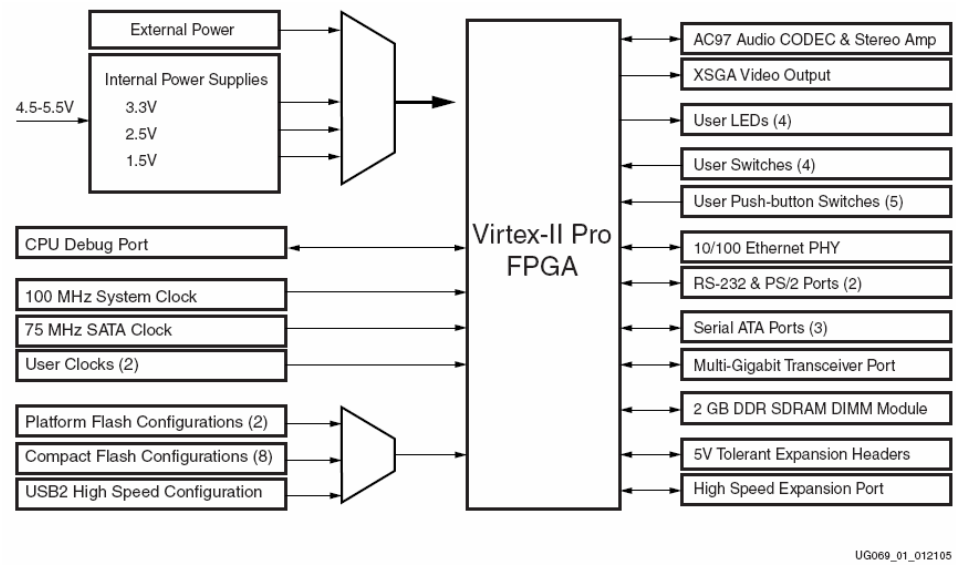


Figure 53. System Core Diagram for the Development Board

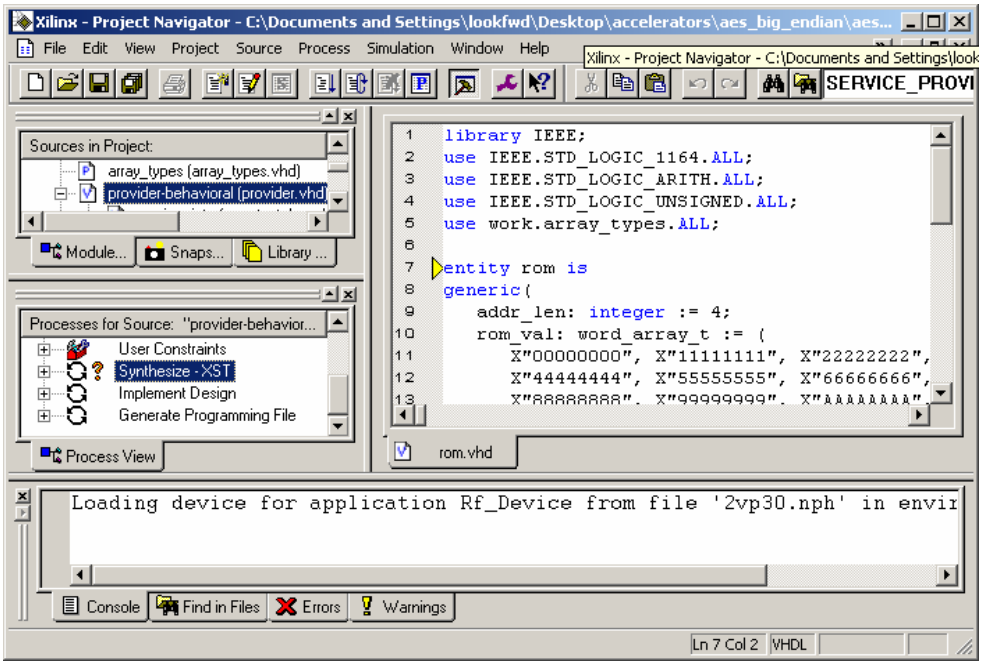


Figure 54. Xilinx ISE development environment

Software development of simple embedded applications is also supported within the same IDE with the PowerPC/Microblaze compilers, debuggers, simulators, in circuit debuggers and most importantly the OS and libraries collection (see [114]). XPS builds the platform which is then implemented with Xilinx's traditional design flow. The only difference is that after placement and routing, an initialization of the Block RAMs with the compiled source code takes place. XPS allows a certain degree of flexibility. Hardware modules can be imported and exported from other tools including ISE and Synplicity [115]. Software can be written using external IDEs like Eclipse which is installed by default with XPS. A complete overview of XPS's features is given in its help and documentation [116].

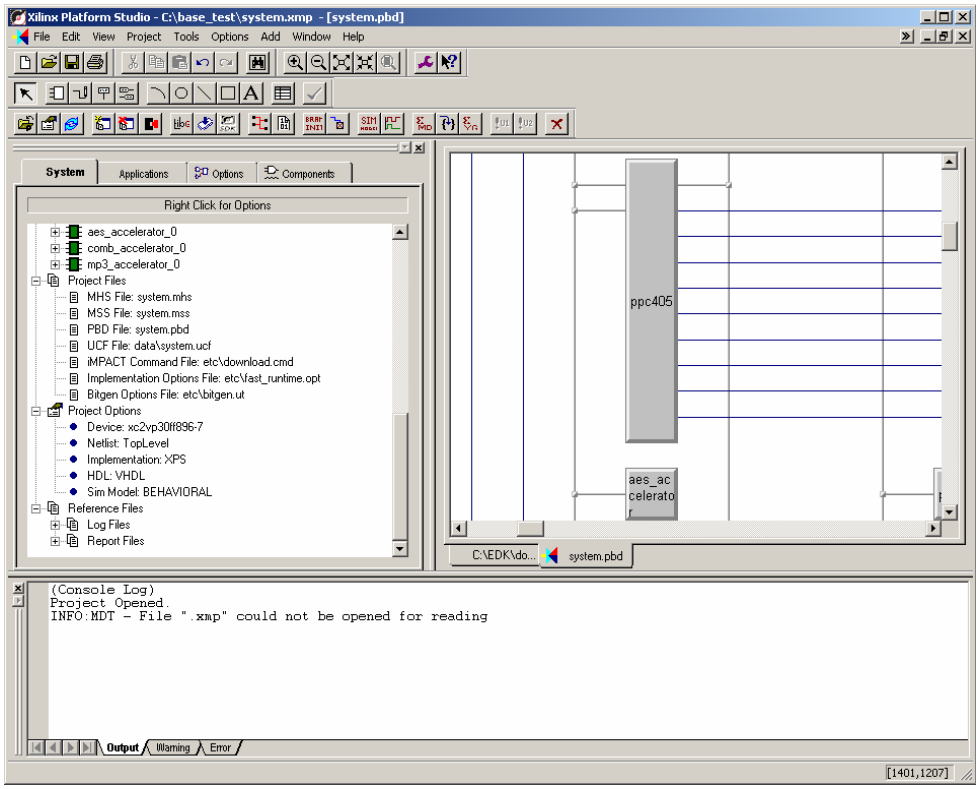


Figure 55. Xilinx XPS development environment

A.2 Dynamic reconfiguration flow

Unfortunately dynamic reconfiguration flow is not supported by current tools and must be done manually by modifying and running batch files. It is described in detail in an application note [117] and is based to a restricted subset of the modular design flow described in [118]. Each module is being synthesized, mapped, placed and routed independently but is forced to fit into certain slice columns. Then the components of the system are being merged at bitstream level. Much help is provided by various tutorials like [119, 120].

If one wants to apply the dynamic reconfiguration flow on platforms generated with Xilinx's XPS there are several large problems. First of all the I/O pins used by the system necessarily span the whole FPGA's area on most development boards. The input bitstream has to be fetched from the RAM whose I/Os are usually in the left side and is written to the ICAP device which lies in the right side. This means that we need wires running over the width of the design that must be active during reconfiguration. Xilinx states explicitly that all the connections that span through a reconfigurable slice should be considered inactive during reconfiguration. Hopefully this isn't true if one uses hard macros and significantly modified flow as demonstrated in [121]. The second problem is that XPS designs can't fit directly into the dynamic reconfiguration flow because they use components like the DCM module which lies outside reconfigurable slices' area. Hopefully there is a workaround for this case as well as described in detail at [122]. Note that both these papers were written in 2006 although the tools are available for more than 5 years.

We went through the dynamic reconfiguration flow ourselves. Simple designs were easily and successfully made using bus macros. We found difficulties on constraining more complex designs into slice columns. Wires were running outside our predefined

borders or routing tools were terminating telling that the design is impossible to be mapped or routed. We moved to XPS project level creating a PowerPC application. Then we used a simple reconfiguration bitstream created with the “simple” (second) from the two design flows described in [117] by modifying manually the design with Xilinx FPGA editor. The bitstream (.bit file) was converted to a variable for use within software by using the script files found on the Xilinx’s application note [123]. Reconfiguration was realized by using the ICAP module and the HWICAP driver documented in [124]. We achieved dynamic reconfiguration successfully with this stream on a running PowerPC system. The code can be found in Appendix C.1 and uses only three API calls for reconfiguration. It must be noted that connections between RAM and PowerPC were running over the slice under configuration and were used during reconfiguration and the design worked perfectly.

A Linux Device Driver for HWICAP doesn’t exist yet but Xilinx promises that will have one soon and very well integrated into the Linux Operating System [110]. We used our successfully tested version of HWICAP driver to our Linux Device Driver. We did not create real reconfiguration streams because in that case we would have to devote considerable but most importantly unpredictable amount of time dealing with very low-level problems which are out of the subject of our work. We used instead a Service Provider able to provide two Core Services that is being considered as reconfigurable from the Service Broker. We simulate reconfiguration by including a reconfiguration delay during which the component is unavailable. This simulates accurately the process of reconfiguration with the only exception of not including the bandwidth required for transferring the reconfiguration stream from RAM to ICAP. HWICAP is an OPB peripheral and thus slow enough to make this bandwidth negligible.

A.3 Montavista Linux

MontaVista Linux is provided by MontaVista. It’s a port of Linux for the Xilinx ml310 board. With the precious help of Jamie [125] we can create a gcc cross compiler for PowerPC, download the source code for this Linux, compile it and load it along with a complete file system to the XUP board that we have. She also shows how to implement some simple device drivers for this Linux [126]. We need and use Linux because it provides us with a basic level of functionality like file and task management in order to execute our demonstration applications. Because it’s a multitasking operating system, we can see the effects of running multiple instances of our applications that reveal the functionality of the Service Broker and reconfiguration functionality.

Appendix B. Hardware entities

B.1 Service Interface

```
package array_types is
	-- CORE TYPE DEFINITIONS
	type word_array_t is array (integer range <>) of
std_logic_vector(31 downto 0);
	type VARIABLE_LENGTHS_t is array(integer range <>) of integer;
	type SERVICE_PROVIDER_DATA_t is record
		SERVICE_ID: integer;
		SERVICE_OUT_PARAMS: integer;
		SERVICE_FAULT_TOLERANCE: boolean;
	end record;
	type SERVICE_PROVIDERS_DATA_t is array(integer range <>) of
SERVICE_PROVIDER_DATA_t;
	type SERVICE_PROVIDER_VARS_t is array(integer range <>, integer
range <>) of integer;

	-- CORE CONSTANT DEFINITIONS
	constant SERVICES_COUNT:integer := 2;
	constant MAX_INPUT_VARS:integer := 4;
	constant MAX_OUTPUT_VARS:integer := 3;
	constant MAX_VAR_WITH_FT: integer := 2;

	constant VARIABLE_LENGTHS_IN : VARIABLE_LENGTHS_t(0 to
MAX_INPUT_VARS-1) := (16, 16, 16, 16);
	constant VARIABLE_LENGTHS_OUT : VARIABLE_LENGTHS_t(0 to
MAX_OUTPUT_VARS-1) := (16, 16, 16);

	constant SERVICE_PROVIDERS_DATA: SERVICE_PROVIDERS_DATA_t(0 to
SERVICES_COUNT-1) := (
		(SERVICE_ID=>0, SERVICE_OUT_PARAMS=>2,
SERVICE_FAULT_TOLERANCE=>TRUE),
		(SERVICE_ID=>1, SERVICE_OUT_PARAMS=>3,
SERVICE_FAULT_TOLERANCE=>FALSE)
	);
	constant SERVICE_PROVIDER_VARS: SERVICE_PROVIDER_VARS_t(0 to
SERVICES_COUNT-1, 0 to MAX_OUTPUT_VARS-1) := (
		(16, 16, 0),
		(16, 16, 16)
	);

	function GET_OUT_PARAMS(sid: integer) return integer;
	function GET_VAR_LENGTH(sid: integer; var: integer) return
integer;
	function GET_FAULT_TOLERANCE(sid: integer) return boolean;
end array_types;
```

B.2 Service Interface

```
entity service_interface is
	Port (
		-- bus interface
		clk : in std_logic;
		reset : in std_logic;
		SBSR_out : out std_logic_vector(31 downto 0);
		SBDR_out : out std_logic_vector(31 downto 0);
```

```
            SBDR_in : in std_logic_vector(31 downto 0);
            SBDR_re : in std_logic;
            SBDR_ac : out std_logic;
            SBDR_we : in std_logic;
            -- service interface
            srv_init: out std_logic;
            srv_go: out std_logic;
            srv_done: in std_logic;
            srv_data_out: out std_logic_vector(31 downto 0);
            srv_data_in: in std_logic_vector(31 downto 0);
            srv_data_in_ac: in std_logic;
            srv_current_variable: out std_logic_vector(7 downto 0);
            srv_current_word: out std_logic_vector(31 downto 0);
            srv_return_parameters: in std_logic_vector(7 downto 0);
            srv_var_size: in std_logic_vector(31 downto 0);
            srv_var_size_ac: in std_logic;
            srv_csum: in std_logic_vector(31 downto 0);
            srv_csum_request: out std_logic;
            srv_csum_ac: in std_logic;
            srv_service_id: out std_logic_vector(15 downto 0);
            srv_variable_valid: out std_logic
      );
end service_interface;
```

B.3 Default Variable Manager

```
entity default_variable_manager is
      port (
            -- Global
      clk : in std_logic;
      reset : in std_logic;
            -- Communication with the interface
            init: in std_logic;
            go: in std_logic;
            done: out std_logic;
            data_out: in std_logic_vector(31 downto 0);
            data_in: out std_logic_vector(31 downto 0);
            data_in_ac: out std_logic;
            current_variable: in std_logic_vector(7 downto 0);
            current_word: in std_logic_vector(31 downto 0);
            return_parameters: out std_logic_vector(7 downto 0);
            var_size: out std_logic_vector(31 downto 0);
            var_size_ac: out std_logic;
            csum: out std_logic_vector(31 downto 0);
            csum_request: in std_logic;
            csum_ac: out std_logic;
            service_id: in std_logic_vector(15 downto 0);
            variable_valid: in std_logic;
            -- Communication with the services
            proc_go: out std_logic;
            proc_done: in std_logic;
            proc_service_id: out std_logic_vector(15 downto 0);
            proc_in_word: in word_array_t(0 to MAX_INPUT_VARS-1);
            proc_in_values: out word_array_t(0 to MAX_INPUT_VARS-1);
            proc_out_word: in word_array_t(0 to MAX_OUTPUT_VARS-1);
            proc_out_we: in std_logic_vector(0 to MAX_OUTPUT_VARS-1);
            proc_out_values: in word_array_t(0 to MAX_OUTPUT_VARS-1);
            proc_out_crc32: in word_array_t(0 to MAX_VAR_WITH_FT-1)
      );
```

```
end default_variable_manager;
```

B.4 Default Services

```
entity default_services is
      port (
            clk: in std_logic;
            reset: in std_logic;
            go: in std_logic;
            done: out std_logic;
            service_id: in std_logic_vector(15 downto 0);
            in_word: out word_array_t(0 to MAX_INPUT_VARS-1);
            in_values: in word_array_t(0 to MAX_INPUT_VARS-1);
            out_word: out word_array_t(0 to MAX_OUTPUT_VARS-1);
            out_we: out std_logic_vector(0 to MAX_OUTPUT_VARS-1);
            out_values: out word_array_t(0 to MAX_OUTPUT_VARS-1);
            out_crc32: out word_array_t(0 to MAX_VAR_WITH_FT-1)
      );
end entity;
```

Appendix C. Source Code

C.1 Reconfiguration through HWICAP

```
#include "xparameters.h"
#include "xgpio_l.h"
#include "xhwicap.h"

unsigned char stream[5948] = {0, 9, 15, 240, 15, 240, 15, 240, 15,
240, 0, 0, 1, 97, 0, 13, 115, 121, 115, …};

unsigned char strea2[2620] = {0, 9, 15, 240, 15, 240, 15, 240, 15,
240, 0, 0, 1, 97, 0, 11, 115, 121, 115, …};

int main (void) {
   int i=0, j=0; volatile int delay=0; int numTimes = 5; char c;
   XHwIcap hIC;

   XStatus status = XHwIcap_Initialize(
      &hIC,
      XPAR_OPB_HWICAP_0_DEVICE_ID,
      XHI_XC2VP30
   );
   if (status != XST_SUCCESS) {
      xil_printf("ICAP Init failed %x status %x\r\n",
         hIC.DeviceIdCode, status);
      exit(1);
   }

   XHwIcap_CommandDesync(&hIC);

   while (1) {
        putchar(c = getchar());
        if (c == '1' | c == '2') {
               unsigned char * ucPtr = c == '1' ? stream : strea2;
               int size = c == '1' ? 1487 : 655;
               switch (XHwIcap_SetConfiguration(&hIC, ucPtr, size)) {
```

```
                  case XST_SUCCESS:
                        xil_printf("XST_SUCCESS\r\n");
                        break;
                  case XST_BUFFER_TOO_SMALL:
                        xil_printf("XST BUFFER TOO SMALL\r\n");
                        break;
                  case XST_INVALID_PARAM:
                        xil_printf("XST INVALID PARAM\r\n");
                        break;
               }
        }

        XGpio_mSetDataDirection(XPAR_LEDS_8BIT_BASEADDR,
              1, 0x00000000);

      j = 1;
      for(i=0; i<8; i++) {
         XGpio_mSetDataReg(XPAR_LEDS_8BIT_BASEADDR, 1, j);
         j = j << 1;
         for (delay=0; delay<100000; delay++);
      }
      j = 1;
      j = ~j;
      for(i=0; i<8; i++) {
         XGpio_mSetDataReg(XPAR_LEDS_8BIT_BASEADDR, 1, j);
         j = j << 1;
         for (delay=0; delay<100000; delay++);
      }
   }
   return 0;
}
```

C.2 rijndaelEncrypt AES encryption function

```
int    rijndaelEncrypt    (word8    a[16],    word8    b[16],    word8
rk[MAXROUNDS+1][4][4])
{
      /* Encryption of one block.
       */
      int r;
   word8 temp[4][4];

    *((word32*)temp[0]) = *((word32*)a) ^ *((word32*)rk[0][0]);
    *((word32*)temp[1]) = *((word32*)(a+4)) ^ *((word32*)rk[0][1]);
    *((word32*)temp[2]) = *((word32*)(a+8)) ^ *((word32*)rk[0][2]);
    *((word32*)temp[3]) = *((word32*)(a+12)) ^ *((word32*)rk[0][3]);
    *((word32*)b) = *((word32*)T1[temp[0][0]])
          ^ *((word32*)T2[temp[1][1]])
          ^ *((word32*)T3[temp[2][2]])
          ^ *((word32*)T4[temp[3][3]]);
    *((word32*)(b+4)) = *((word32*)T1[temp[1][0]])
          ^ *((word32*)T2[temp[2][1]])
          ^ *((word32*)T3[temp[3][2]])
          ^ *((word32*)T4[temp[0][3]]);
    *((word32*)(b+8)) = *((word32*)T1[temp[2][0]])
          ^ *((word32*)T2[temp[3][1]])
          ^ *((word32*)T3[temp[0][2]])
          ^ *((word32*)T4[temp[1][3]]);
    *((word32*)(b+12)) = *((word32*)T1[temp[3][0]])
          ^ *((word32*)T2[temp[0][1]])
```

```
           ^ *((word32*)T3[temp[1][2]])
           ^ *((word32*)T4[temp[2][3]]);
   for(r = 1; r < ROUNDS-1; r++) {
            *((word32*)temp[0])        =        *((word32*)b)        ^
*((word32*)rk[r][0]);
            *((word32*)temp[1])      =      *((word32*)(b+4))      ^
*((word32*)rk[r][1]);
            *((word32*)temp[2])      =      *((word32*)(b+8))      ^
*((word32*)rk[r][2]);
            *((word32*)temp[3])      =      *((word32*)(b+12))      ^
*((word32*)rk[r][3]);
   *((word32*)b) = *((word32*)T1[temp[0][0]])
           ^ *((word32*)T2[temp[1][1]])
           ^ *((word32*)T3[temp[2][2]])
           ^ *((word32*)T4[temp[3][3]]);
   *((word32*)(b+4)) = *((word32*)T1[temp[1][0]])
           ^ *((word32*)T2[temp[2][1]])
           ^ *((word32*)T3[temp[3][2]])
           ^ *((word32*)T4[temp[0][3]]);
   *((word32*)(b+8)) = *((word32*)T1[temp[2][0]])
           ^ *((word32*)T2[temp[3][1]])
           ^ *((word32*)T3[temp[0][2]])
           ^ *((word32*)T4[temp[1][3]]);
   *((word32*)(b+12)) = *((word32*)T1[temp[3][0]])
           ^ *((word32*)T2[temp[0][1]])
           ^ *((word32*)T3[temp[1][2]])
           ^ *((word32*)T4[temp[2][3]]);
   }
   /* last round is special */
      *((word32*)temp[0])  =  *((word32*)b)  ^  *((word32*)rk[ROUNDS-
1][0]);
      *((word32*)temp[1]) = *((word32*)(b+4)) ^ *((word32*)rk[ROUNDS-
1][1]);
      *((word32*)temp[2]) = *((word32*)(b+8)) ^ *((word32*)rk[ROUNDS-
1][2]);
      *((word32*)temp[3])        =        *((word32*)(b+12))        ^
*((word32*)rk[ROUNDS-1][3]);
   b[0] = T1[temp[0][0]][1];
   b[1] = T1[temp[1][1]][1];
   b[2] = T1[temp[2][2]][1];
   b[3] = T1[temp[3][3]][1];
   b[4] = T1[temp[1][0]][1];
   b[5] = T1[temp[2][1]][1];
   b[6] = T1[temp[3][2]][1];
   b[7] = T1[temp[0][3]][1];
   b[8] = T1[temp[2][0]][1];
   b[9] = T1[temp[3][1]][1];
   b[10] = T1[temp[0][2]][1];
   b[11] = T1[temp[1][3]][1];
   b[12] = T1[temp[3][0]][1];
   b[13] = T1[temp[0][1]][1];
   b[14] = T1[temp[1][2]][1];
   b[15] = T1[temp[2][3]][1];
      *((word32*)b) ^= *((word32*)rk[ROUNDS][0]);
      *((word32*)(b+4)) ^= *((word32*)rk[ROUNDS][1]);
      *((word32*)(b+8)) ^= *((word32*)rk[ROUNDS][2]);
      *((word32*)(b+12)) ^= *((word32*)rk[ROUNDS][3]);

      return 0;
}
```

C.3 synth_full mp3 decoding function

```
static
void synth_full(struct mad_synth *synth, struct mad_frame const
*frame,
            unsigned int nch, unsigned int ns)
{
  unsigned int phase, ch, s, sb, pe, po;
  mad_fixed_t *pcm1, *pcm2, (*filter)[2][2][16][8];
  mad_fixed_t const (*sbsample)[36][32];
  register mad_fixed_t (*fe)[8], (*fx)[8], (*fo)[8];
  register mad_fixed_t const (*Dptr)[32], *ptr;
  register mad_fixed64hi_t hi;
  register mad_fixed64lo_t lo;

  for (ch = 0; ch < nch; ++ch) {
    sbsample = &frame->sbsample[ch];
    filter   = &synth->filter[ch];
    phase    = synth->phase;
    pcm1     = synth->pcm.samples[ch];

    for (s = 0; s < ns; ++s) {
      dct32((*sbsample)[s], phase >> 1,
          (*filter)[0][phase & 1], (*filter)[1][phase & 1]);

      pe = phase & ~1;
      po = ((phase - 1) & 0xf) | 1;

      /* calculate 32 samples */

      fe = &(*filter)[0][ phase & 1][0];
      fx = &(*filter)[0][~phase & 1][0];
      fo = &(*filter)[1][~phase & 1][0];

      Dptr = &D[0];

      ptr = *Dptr + po;
      ML0(hi, lo, (*fx)[0], ptr[ 0]);
      MLA(hi, lo, (*fx)[1], ptr[14]);
      MLA(hi, lo, (*fx)[2], ptr[12]);
      MLA(hi, lo, (*fx)[3], ptr[10]);
      MLA(hi, lo, (*fx)[4], ptr[ 8]);
      MLA(hi, lo, (*fx)[5], ptr[ 6]);
      MLA(hi, lo, (*fx)[6], ptr[ 4]);
      MLA(hi, lo, (*fx)[7], ptr[ 2]);
      MLN(hi, lo);

      ptr = *Dptr + pe;
      MLA(hi, lo, (*fe)[0], ptr[ 0]);
      MLA(hi, lo, (*fe)[1], ptr[14]);
      MLA(hi, lo, (*fe)[2], ptr[12]);
      MLA(hi, lo, (*fe)[3], ptr[10]);
      MLA(hi, lo, (*fe)[4], ptr[ 8]);
      MLA(hi, lo, (*fe)[5], ptr[ 6]);
      MLA(hi, lo, (*fe)[6], ptr[ 4]);
      MLA(hi, lo, (*fe)[7], ptr[ 2]);

      *pcm1++ = SHIFT(MLZ(hi, lo));

      pcm2 = pcm1 + 30;
```

```
      for (sb = 1; sb < 16; ++sb) {
      ++fe;
      ++Dptr;

      /* D[32 - sb][i] == -D[sb][31 - i] */

      ptr = *Dptr + po;
      ML0(hi, lo, (*fo)[0], ptr[ 0]);
      MLA(hi, lo, (*fo)[1], ptr[14]);
      MLA(hi, lo, (*fo)[2], ptr[12]);
      MLA(hi, lo, (*fo)[3], ptr[10]);
      MLA(hi, lo, (*fo)[4], ptr[ 8]);
      MLA(hi, lo, (*fo)[5], ptr[ 6]);
      MLA(hi, lo, (*fo)[6], ptr[ 4]);
      MLA(hi, lo, (*fo)[7], ptr[ 2]);
      MLN(hi, lo);

      ptr = *Dptr + pe;
      MLA(hi, lo, (*fe)[7], ptr[ 2]);
      MLA(hi, lo, (*fe)[6], ptr[ 4]);
      MLA(hi, lo, (*fe)[5], ptr[ 6]);
      MLA(hi, lo, (*fe)[4], ptr[ 8]);
      MLA(hi, lo, (*fe)[3], ptr[10]);
      MLA(hi, lo, (*fe)[2], ptr[12]);
      MLA(hi, lo, (*fe)[1], ptr[14]);
      MLA(hi, lo, (*fe)[0], ptr[ 0]);

      *pcm1++ = SHIFT(MLZ(hi, lo));

      ptr = *Dptr - pe;
      ML0(hi, lo, (*fe)[0], ptr[31 - 16]);
      MLA(hi, lo, (*fe)[1], ptr[31 - 14]);
      MLA(hi, lo, (*fe)[2], ptr[31 - 12]);
      MLA(hi, lo, (*fe)[3], ptr[31 - 10]);
      MLA(hi, lo, (*fe)[4], ptr[31 -  8]);
      MLA(hi, lo, (*fe)[5], ptr[31 -  6]);
      MLA(hi, lo, (*fe)[6], ptr[31 -  4]);
      MLA(hi, lo, (*fe)[7], ptr[31 -  2]);

      ptr = *Dptr - po;
      MLA(hi, lo, (*fo)[7], ptr[31 -  2]);
      MLA(hi, lo, (*fo)[6], ptr[31 -  4]);
      MLA(hi, lo, (*fo)[5], ptr[31 -  6]);
      MLA(hi, lo, (*fo)[4], ptr[31 -  8]);
      MLA(hi, lo, (*fo)[3], ptr[31 - 10]);
      MLA(hi, lo, (*fo)[2], ptr[31 - 12]);
      MLA(hi, lo, (*fo)[1], ptr[31 - 14]);
      MLA(hi, lo, (*fo)[0], ptr[31 - 16]);

      *pcm2-- = SHIFT(MLZ(hi, lo));

      ++fo;
      }

      ++Dptr;

      ptr = *Dptr + po;
      ML0(hi, lo, (*fo)[0], ptr[ 0]);
      MLA(hi, lo, (*fo)[1], ptr[14]);
      MLA(hi, lo, (*fo)[2], ptr[12]);
```

```
      MLA(hi, lo, (*fo)[3], ptr[10]);
      MLA(hi, lo, (*fo)[4], ptr[ 8]);
      MLA(hi, lo, (*fo)[5], ptr[ 6]);
      MLA(hi, lo, (*fo)[6], ptr[ 4]);
      MLA(hi, lo, (*fo)[7], ptr[ 2]);

      *pcm1 = SHIFT(-MLZ(hi, lo));
      pcm1 += 16;

      phase = (phase + 1) % 16;
    }
  }
}
# endif
```

Appendix D. Various topics

D.1 The checksum

When we talk about checksums, the first thing that comes to mind is LFSR. Unfortunately this robust solution has expensive software implementation. A 32 bit LFSR checksum can be done with the following routine:

```
output = input + (output << 1) + ( \
      (output & (1<<1) ? 1 : 0) ^  \
      (output & (1<<5) ? 1 : 0) ^  \
      (output & (1<<6) ? 1 : 0) ^  \
      (output & (1<<31) ? 1 : 0)   \
);
```

We can see that it includes a lot of unnecessary 'if', 32-bit shift, 'and' and xor operations. With some profiling we calculate that it takes almost 40 clock cycles per iteration for Pentium architecture. In hardware, its implementation is straightforward.

Instead of LFSR we use the following compression method to calculate checksum:

```
output += input;
output = (output <<1) + (output & 0x80000000?1:0);
```

This takes 12 clock cycles per iteration for Pentium, more than three times faster (both compilations used –O3). In hardware this is nothing more than an adder and a rotating shift register.

```
acc <= output + input;
process(clk) begin
      if (rising_edge(clk)) then
            output <= acc(30 downto 0) & acc(31);
      end if;
end process;
```

Obviously it's fast and it consumes very few resources.

D.2 I/O operation efficiency

On an standalone XPS software project (no Linux) we created a program that was making 10.000 reads and writes on a register in order to benchmark their performance and choose the best for our implementation.

By calling the code from the original XIo_Out32 and XIo_In32 functions we got the following profiles:

> Writes: 1680039 clock cycles => 168 c/write
> Reads: 1530060 clock cycles => 153 c/write

After enabling the instruction cache we faced a 268% improvement on reads and 466% improvement on writes.

> Writes: 360091 clock cycles => 36 c/write
> Reads: 570089 clock cycles => 57 c/write

After enabling and the data cache we faced another 172% improvement on reads which gave us equal read and write times.

> Writes: 360046 clock cycles => 36 c/write
> Reads: 330062 clock cycles => 33 c/write

An assembly instruction (eieio) was used to put a barrier that ensures that all the I/O opretions complete in sequence. This was executed after our I/O operation making the whole implementation slower because on an actual run the following was happening:

```
stw %0, 0(%1); eieio – Waits
-- loop code
stw %0, 0(%1); eieio – Waits again
```

We modified the code slightly by putting the eieio instruction before the I/O operation. This way the branch overhead was coming at free under the I/O synchronization operation:

```
eieio; stw %0, 0(%1)
-- loop code doesn't provide an overhead
eieio ; stw %0, 0(%1) – Waits again
```

After changing the order of write and eieio in batch_write (10% improvement).

> Writes: 300047 clock cycles => 30 c/write
> Reads: 300050 clock cycles => 30 c/write

Further loop unrolling results, inlining assembly and other tricks didn't provide any further acceleration. Because the CPU was running on 300MHz the numbers above actually mean that a single read/write completes in 10 bus clock cycles.

D.3 XPS project debug and time traces

Testing AES_ACCELERATOR_0: AES_BARE
sending: 555, computing: 118, receiving: 70, total: 743
var[1] = {1CB0CBD9, C9304FA8, AC711D92, 7CE2E30F}
checksum = {06084922}

Testing AES_ACCELERATOR_0: AES_FULL
sending: 427, computing: 118, receiving: 168, total: 713
var[1] = {FC1B9283, 533D5E1E, 93CD84C3, B09B3308}
checksum = {06084922}

Testing COMB_ACCELERATOR_0: AES_FULL
sending: 427, computing: 118, receiving: 153, total: 698
var[1] = {FC1B9283, 533D5E1E, 93CD84C3, B09B3308}
checksum = {06084922}

Testing COMB_ACCELERATOR_0: AES_FULL

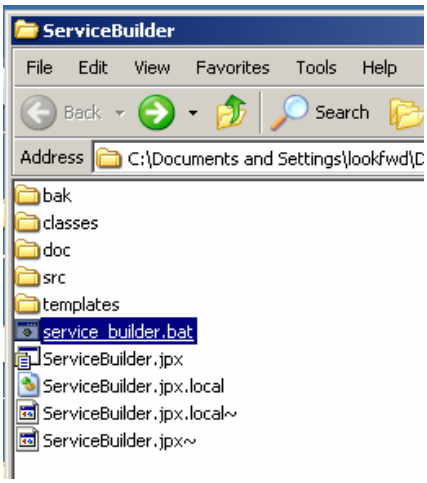
```
sending: 427, computing: 118, receiving: 153, total: 698
var[1] = {FC1B9283, 533D5E1E, 93CD84C3, B09B3308}
checksum = {06084922}

Testing software AES process
time: 1824
var[1] = {FC1B9283, 533D5E1E, 93CD84C3, B09B3308}
checksum = {06084922}
```

```
Testing MP3_ACCELERATOR_0: MP3_NO
sending: 215, computing: 34, receiving: 122, total: 371
var[1] = {00009A3A, FFFF5889}

Testing COMB_ACCELERATOR_0: MP3_NO
sending: 171, computing: 34, receiving: 101, total: 306
var[1] = {00009A3A, FFFF5889}

Testing COMB_ACCELERATOR_0: MP3_NO
sending: 171, computing: 34, receiving: 101, total: 306
var[1] = {00009A3A, FFFF5889}

Testing software MP3 process
time: 1056
var[1] = {00009A3A, FFFF5889}
```

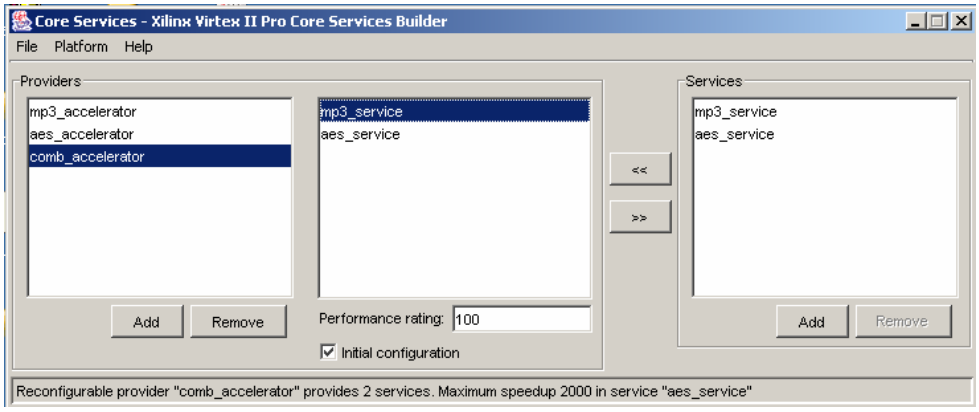
D.4 A quick tutorial in Core Services

We provide this quick tutorial in order to help you get started with Core Services.

At the beginning you should start the Service Builder application by clicking on the appropriate bat file.

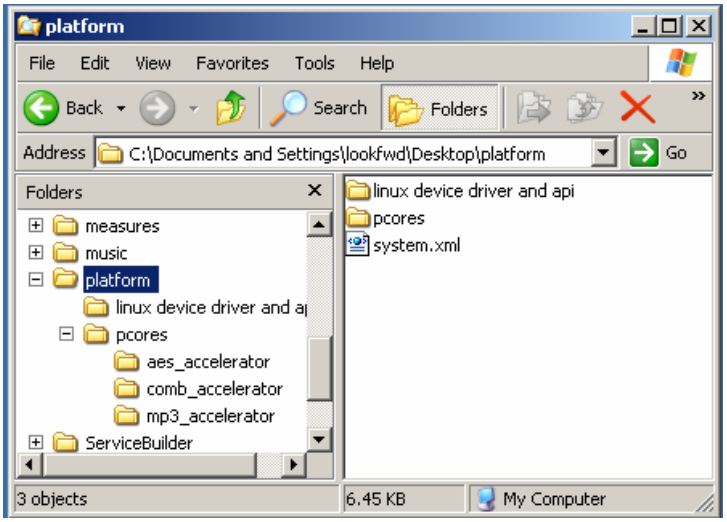


Service Builder's user interface initializes and from here you can customize Core Services' platform by using its menus and dialogs. Useful information about performance can be found in the status bar.



Once the customization is complete and you are satisfied with the platform you select Generate from the Platform menu. It asks you for a folder to place the files and Core Service's files get generated automatically.

You can see in the following figure the directory hierarchy that gets generated. A directory gets generated for each accelerator on the pcores directory. All the software components lie in the "linux device driver and api" directory. Service Builder automatically saves a copy of the system configuration in that folder in order to be able to open it and find out what current configuration provides.



Then hardware generation using XPS takes place. You can find a step-by-step video presentation on my web site lookfwd.doitforme.gr/projects on this subject. An overview is provided here.

1. Create a new project by using Base System Builder. The project should represent your computational needs and also don't occupy the whole FPGA area in order to leave space for accelerator's hardware. You may need to consult Jamie's page [125] to find details on a Base System that can run Linux. You may find that a Ethernet MAC is not necessary for simple experiments because it takes considerable time to synthesize.

2. Import each design from the pcores folder by using the Import Peripheral Wizard (Tools > Create/Import Peripheral). Use the “Add Library” command to resolve any library dependencies.
3. Use Add/Edit Cores dialog (Project > Add/Edit Cores) to connect accelerating components to the system. Do the appropriate bus connections and Generate Addresses.
4. Do the rest of the flow for hardware generation as described in Jamie’s page [125].

At the end of this process you will have in the implementation folder of your project a download.bit file that contains the bitstream for you application. Then you have to create the software by following the software generation flow described in Jamie’s page [125]. You generate a cross compiler for PowerPC and a MontaVista linux for the platform.

Then you take the files from the “linux device driver and api” for cross compilation. The following Makefile was used for compiling our driver and might be found to be useful:

```
KERNELDIR=/root/Desktop/cross/linuxppc_2_4_devel

include $(KERNELDIR)/.config

CC = powerpc-405-linux-gnu-gcc
LD = powerpc-405-linux-gnu-ld
CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include \
    -I$(KERNELDIR)/arch/ppc \
    -O2 -Wall

ifdef CONFIG_SMP
  CFLAGS += -D__SMP__ -DSMP -Wall
endif

all: cs.o ioctltest

ioctltest:  ioctltest.c  core_services.c  core_services.h  csdriver.h
aesdefault.c mp3default.c
      ${CC}   -Wall   -O   ioctltest.c   core_services.c   aesdefault.c
mp3default.c -o ioctltest
csdriver.o:   csdriver.c reconf.c reconf.h csdriver.h low_level_io.h
platform.h
platform.o: platform.c platform.h

cs.o: platform.o csdriver.o
      $(LD) -r -o $@ platform.o csdriver.o


clean:
      rm -f csdriver.o ioctltest cs.o platform.o
```

Inside core_services.h you can find macros that wrap the core services API and provide a plain function API for our Core Services. You can see in the following section these functions for our case:

```
#define aesService(a,b,rk, redundancy) ({ \
unsigned int *aes_inargv[] = {a, rk}; \
unsigned int aes_inargc[] = {4, 44}; \
unsigned int *aes_outv[] = {b}; \
```

```
unsigned int outargc[1]; \
 highLevelCallService(aesdefault,    redundancy,    AES_SERVICE,    2,
aes_inargv, aes_inargc, aes_outv, outargc); \
})

#define mp3Service(parm, ret) ({ \
unsigned int *mp3_inargv[] = {parm}; \
unsigned int mp3_inargc[] = {17}; \
unsigned int *mp3_outv[] = [65]; \
unsigned int outargc[1]; \
 highLevelCallService(mp3default,  1,  MP3_SERVICE,  1,  mp3_inargv,
mp3_inargc, mp3_outv, outargc); \
})
```

These can get called from within application code as simply as this:

```
int    rijndaelEncrypt    (word8    a[16],    word8    b[16],    word8
rk[MAXROUNDS+1][4][4]) {
      aesService(a,b,rk, 2);
}

inarg[0] = phase << 5 | 0;
memcpy(&inarg[1], fx, 32);
memcpy(&inarg[9], fe, 32);
mp3Service(inarg, mp3_outv0);
*pcm1++ = mp3_outv0[0];
```

Don't forget to include the Core Services' header file:

```
#include "core_services.h"
```

Then you copy the compiled files to the compact flash and you can load the driver by typing insmod cs.o and create file entries to its processes by using makenod /dev/devX -c 254 0

You may alternatively automate this process with a batch file like this (in our case we have three Service Providers)

```
#!/bin/sh
insmod cs.o
mknod /dev/csbroker c 254 0
mknod /dev/cs0 c 254 1
mknod /dev/cs1 c 254 2
mknod /dev/cs2 c 254 3
```

Finally you can create performance logs on the Xilinx platform by using our "mthrough" utility to measure the throughput like this:

```
cat 02-counterstrike-truth-trt.mp3 | ./aescrypt -k keyfile.txt -s 128
| ./mthrough > /dev/null 2>> aes0.log &
cat 02-counterstrike-truth-trt.mp3 | ./aescrypt -k keyfile.txt -s 128
| ./mthrough > /dev/null 2>> aes1.log &
cat 02-counterstrike-truth-trt.mp3 | ./aescrypt -k keyfile.txt -s 128
| ./mthrough > /dev/null 2>> aes2.log &
./minimad | ./mthrough > /dev/null < 02-counterstrike-truth-trt.mp3
2>> mp30.log &
./minimad | ./mthrough > /dev/null < 02-counterstrike-truth-trt.mp3
2>> mp31.log &
```

```
./minimad | ./mthrough > /dev/null < 02-counterstrike-truth-trt.mp3
2>> mp32.log &
./minimad | ./mthrough > /dev/null < 02-counterstrike-truth-trt.mp3
2>> mp33.log &
```

Some other useful procedure:

Compiling the minimad mp3 player under windows (cygwin)

```
$ configure --disable-shared --enable-profiling
$ make
$ make minimad.exe
$ gcc -Wall -march=i486 -g -O -fforce-mem -fforce-addr -fthread-jumps
-fcse-follow-jumps  -fcse-skip-blocks  -fexpensive-optimizations  -
fregmove -fschedule-insns2 -fstrength-reduce -o minimad.exe minimad.o
version.o fixed.o bit.o timer.o stream.o frame.o synth.o decoder.o
layer12.o layer3.o huffman.o
```

In order to run applications like minimad on linux ppc you have to cross-compile it using the following instructions:

```
./configure –disable-shared –enable-profiling --build=`config.guess`
--target=powerpc-405-linux-gnu --host=powerpc-405-linux-gnu
make && make minimad
```

You may have to modify slightly the Makefiles in order to include core_service' framework's files.

Using the cross compiled gprof:

```
Powerpc-405-linux-gnu-pgrof minimad > log_ppc405.txt
```

Appendix E. Advanced implementation issues

E.1 On networks-on-chip supporting multicasting

The need to support multicasting is a nice feature for NoCs but there are not so many practical applications that seem to exploit it. With Core Services one can use it to gain a significant decrease of communication costs on Core Services that require fault tolerance. If a fault tolerance of n redundant computations per computation is required and n hardware accelerators are readily available, the total communication time on the service requester is decreased to almost $1/n$ of the original. As we discussed in section 3.4 communication cost is not only considerable but can be significantly larger and much less deterministic than the computation cost.

A recent work in multicasting on a mesh network with deadlock freedom is being presented on [127]. We will use their notions for our multicast implementation. It must be noted that there may be extensions to the Æthereal to support multicasting [128] by using scheduling algorithms from tiny-tera [129].

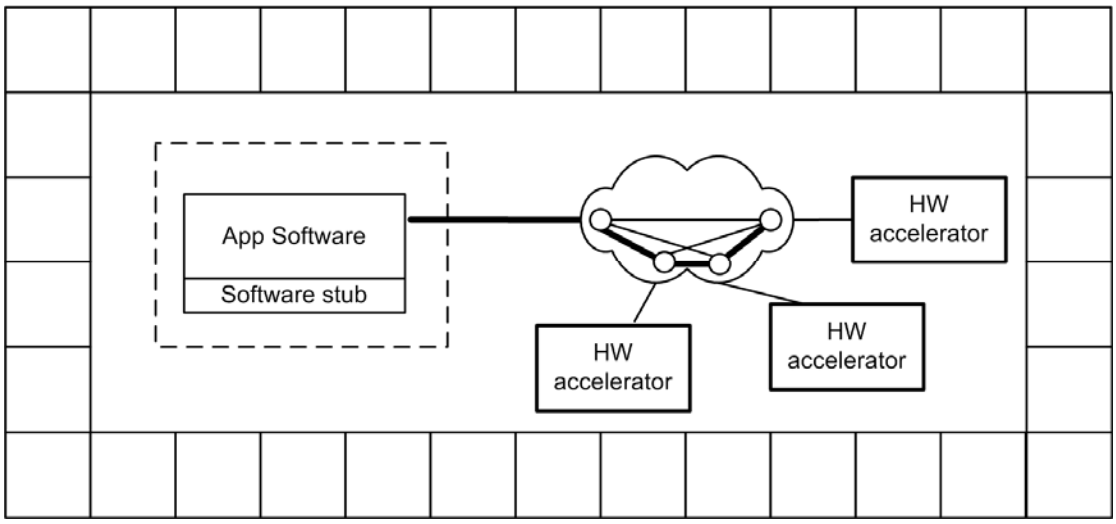


Figure 56. Multicasting scheme of Core Services

The service request phase (Phase I) is identical with the unicast NoC implementation. In service execute (Phase II) the service requester instead of passing the parameters to each service provider in sequence, as in the unicast case, it setups a group by using a packet with “multicast setup” PacketType. If a successful acknowledgment is received then the requester becomes a group master and can send data via the virtual multicast channel (see Figure 56). Then it sends the parameters to all the providers with a single multicast packet (PacketType: “multicast data”) via the dedicated multicast channel (identified by its MultiID). At the end of this transaction, a group release takes place, by sending an appropriate packet (PacketType: “multicast group release”).

Obviously, this implementation of multicasting requires a significant overhead of establishing and the releasing the virtual channel. It gives considerable advantages only on big packets of parameters and increased number of computation redundancy. If more efficient implementation techniques arise, the use of multicasting can give a decrease to almost $1/n$ on the total parameter passing communication cost and the associated energy.

E.2 Interfacing external networks: A case study

A SoC using the Core Services methodology can also be extended to provide and/or consume services from off-chip networks either global like the internet or local on the same board/system. This can be done with special pieces of hardware, bridges, that interface all the aspects of the external network to the internal and vice versa.

For example, we are going to see how a Core Service can invoke a web via a network interface. Suppose that we have a Core Service for an embedded GIS application that takes as parameters the latitude and longitude and returns the temperature at that position. This way the application can show temperature maps over a map. The first version of this product used static weather prediction based on a model of the weather, current date and a temperature sensor. Obviously this was quite inaccurate but the company decided to release it this way and improve it in future version if it received positive feedback. They did it by using a default software implementation running on the same processor as the application.

The positive feedback actually came and it was decided to make the application more accurate by using data from the internet, if an internet connection is available. A processor with 802.11b wireless interface already existed on-chip from the first version to allow the user to get waypoints and tracking information and to provide software and map upgrades.

In order to provide this upgrade, they have just to modify slightly the software of the system. These are the steps:

1. The internet-enabled processor must run a process that checks if internet connection is available (probably it does it already)
2. If internet connection is available it registers itself to the service broker as a provider of the Core Service. If not, then it un-registers itself. It must use an attractive performance parameter in order to outbalance the default implementation.
3. When the processor receives such a service request, it wraps the latitude and longitude in a REST compliant request (see section Chapter 2.) and sends it to a web service provider like weather.gov (www.weather.gov/xml).
4. When it receives the response the processor parses it and retrieves the temperature information. Then it forms a response packet and sends it back to the Core Service request issuer.

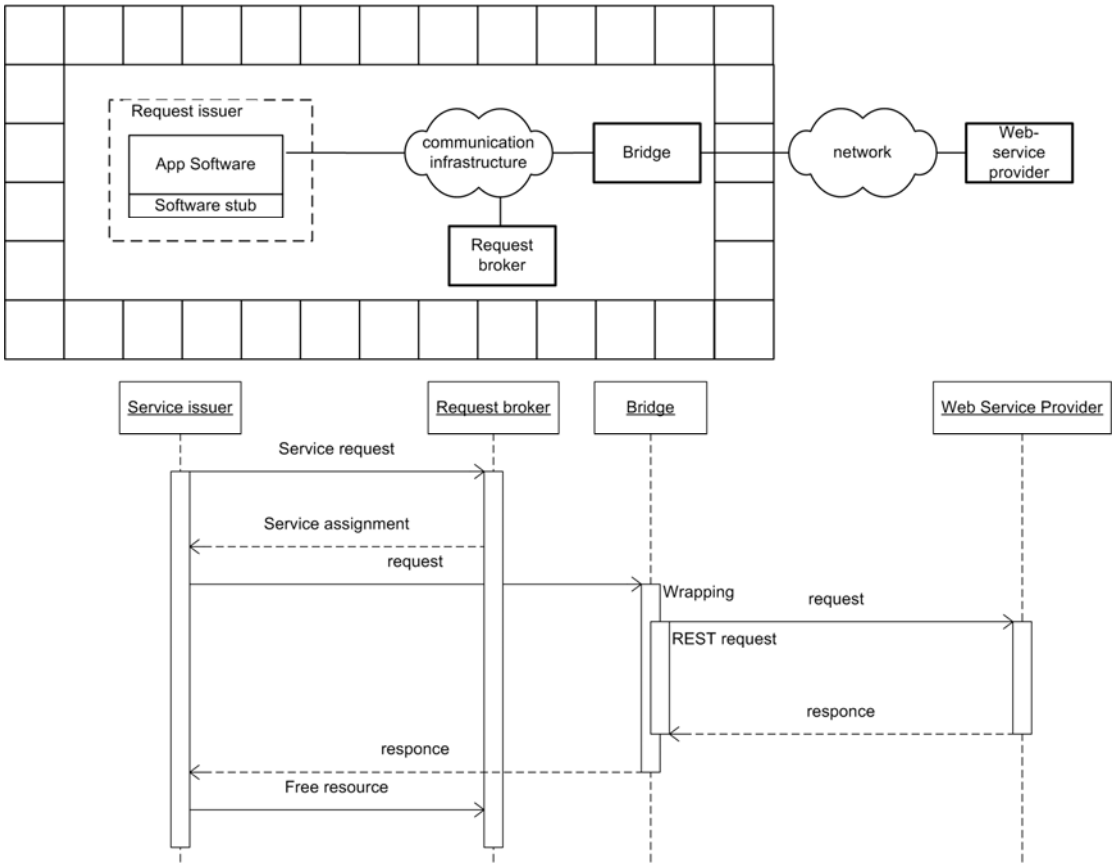


Figure 57. External web service request example

We can see the final process in Figure 57. Unarguably using Core Services makes the system much more flexible. There was no need to change the application software so backwards compatibility is being retained. By changing only platform's firmware we achieved better accuracy. Another advantage is that the communication overhead and the performance are minimized effectively without explicit effort. The on-chip communication infrastructure is being loaded only with the light Core Services messaging and not with any heavyweight xml-http text transferring. All the web text-oriented operations are being performed in the bridge that is probably already optimized for web operations e.g. have better string processing support.

Appendix F. API documentation

As you can see the low level and the high level API are easy to use and reflect directly Core Service's mechanics.

F.1 Low level API

`int getBroker();`

This functionreturns a handle to the Service Broker. This handle can be used to get and return Service Providers' for specific services.

`void returnBroker(int hBroker);`

This function clears the handle to the Service Broker and frees the reference. The reference is invalid after this function.

`int getServiceProviderHandler(int provider_id);`

This function creates a handler for a service provider. The handler is just for accelerating future references to it and it shouldn't be used for services until requested explicitly with a call to the `getServiceProviders` function.

`void returnServiceProviderHandler(int sph);`

This function destroys a handler for a Service Provider. The handler is invalid and can't get used after this function.

`int * initializeCSHandlers(int hBroker);`

This initializes handlers for all the Service Providers available on the system. These handlers are just for accelerating future references to it and shouldn't be used for services until requested explicitly with a call to the `getServiceProviders` function.

`void releaseCSHandlers(int hBroker, int * csHandlers);`

This function releases handler for a Service Providers of initialized by `initializeCSHandlers`.

`int getNumberOfProviders(int hBroker);`

A utility function able to retrieve the number of Service Providers available on the system.

`void getServiceProviders(int hBroker, unsigned int sid, unsigned int redundancy, service_request_t * providers);`

This is the main function used for provider allocation. It takes describes the service ID and the required amount of redundancy for the system. After this call the hardware accelerators (Service Provider) are reserved by the process until explicitly freed with a call to `returnServiceProviders`.

```
void returnServiceProviders(int hBroker, service_request_t *
providers);
```

This function returns the Service Provider from the current process and makes them available for the other processes or processors.

```
unsigned int callService(
      unsigned int hProvider,
      unsigned int sid,
      unsigned int vars,
      unsigned int **inargv,
      unsigned int *inargc,
      unsigned int **outargv,
      unsigned int *outargc,
      unsigned int ftmode,
      unsigned int *checksums
);
```

With this function we call a Core Service on a given service provider. Service call's parameters are passed and result parameters as well as checksums are being provided back.

```
void printResults(unsigned int vars, unsigned int **outargv,
unsigned int *outargc, unsigned int *checksums);
```

This utility function can be used for printing easily the results of a Service Call. It is useful for debugging.

Core Service's IDs:

A service ID is generated for each Core Service in the system. For our application these are:

#define MP3_SERVICE 0
#define AES_SERVICE 1

Fault tolerance modes:

#define FT_MODE_NO_FT 0
#define FT_MODE_FT_FULL 1
#define FT_MODE_FT_BARE 2

Three fault tolerance mode constants are supported. FT_MODE_NO_FT is used when the hardware has no support for fault tolerance. FT_MODE_FT_FULL is used for using fault tolerance and returning the results and the checksums. FT_MODE_FT_BARE FULL is used for using fault tolerance and returning only checksums.

F.2 High level API

A function `template_fun` which provides the default software implementation is defined in the high level API in the following manner:

```
typedef int (*template_fun)(unsigned int vars, unsigned int
**inargv, unsigned int *inargc, unsigned int **outargv,
```

```
unsigned int *outargc, unsigned int ftmode, unsigned int
*checksums);
```

The function used to invoke a Core Service is the following.

```
unsigned int highLevelCallService(
      template_fun fun,
      int redundancy,
      unsigned int sid,
      unsigned int vars,
      unsigned int **inargv,
      unsigned int *inargc,
      unsigned int **outargv,
      unsigned int *outargc
);
```

As we can see the default implementation is provided, the level of fault tolerance (up to 16) and the usual in and out parameters description. Checksums are not provided because they are used internally on this functions and fault tolerance is guaranteed. This function runs the whole process of allocating Service Builder, searching for available Service Providers and checking checsums to provide fault tolerance.

On top of these function application dependent macros are being used to provide a function-level API. They initialize the arguments given to `highLevelCallService` according to platforms' specification. In the case of AES service this has the following form:

```
#define aesService(a,b,rk, redundancy) ({ \
unsigned int *aes_inargv[] = {a, rk}; \
unsigned int aes_inargc[] = {4, 44}; \
unsigned int *aes_outv[] = {b}; \
unsigned int outargc[1]; \
 highLevelCallService(aesdefault, redundancy, AES_SERVICE, 2,
aes_inargv, aes_inargc, aes_outv, outargc); \
})
```