Distributed Object Models for Multi-Processor SoC's, with Application to Low-power Multimedia Wireless Systems

Pierre G. Paulin, Chuck Pilkington, Michel Langevin, Essaid Bensoudane,
Olivier Benny, Damien Lyonnard, Bruno Lavigueur, David Lo
STMicroelectronics, Advanced System Technology
16 Fitzgerald Rd., Nepean, ON, K2H 8R6, Canada
pierre.paulin@st.com

Abstract

This paper summarizes the characteristics of distributed object models used in large-scale distributed software systems. We examine the common subset of requirements for distributed software systems and systems-on-a-chip (SoC), namely: openness, heterogeneity and multiple forms of transparency. We describe the application of these concepts to the emerging class of complex, parallel SoC's, including multiple heterogeneous embedded processors interacting with hardware co-processors and I/O devices. An implementation of this approach is embodied in STMicroelectronics' DSOC (Distributed System Object Component) programming model. The use of this programming model for an architecture exploration of ST's Nomadik mobile multimedia platform is described.

1. Introduction

The current deep submicron technology era presents two opposing challenges: rising SoC platform development costs and shorter product market windows. The rising platform development costs are due to four main sources: the continued rise in gate count, the emergence of deep submicron effects, the rising proportion of embedded software development costs, and finally, rising mask set costs.

As a result, the significant investment to develop the platform – typically between 10M$ and 100M$ for *today's* leading-edge 90nm platforms – requires to maximize the *time-in-market* for a given platform. On the other hand, the consumer-led product cycles imply increasingly shorter *time-to-market* for the applications supported by the platform. Addressing these two conflicting requirements will come from two main directions [1]:

- The development of more flexible platforms that can be used across a wider range of applications and can evolve with short-term market requirements.
- The development of new tools which support the fast and efficient mapping of high-level application descriptions onto these flexible platforms. This implies the development of abstract platform programming models.

2. Challenges of 4G Wireless Systems

Next-generation wireless systems are bound to these same commercial realities. Moreover, competition is extremely high which causes extreme pressure on costs. Differentiation is increasingly coming from the ability to offer distinctive functionality quickly. Multimedia functions such as enhanced audio, video recording and playback, digital still imaging and 3D graphics are emerging as strong differentiators in this market.

Providing these multimedia functions at low power requires the effective use of parallelism. The use of multiple low-frequency, simple processors is a more power effective means of delivering a specific MIPS performance target than a single, high-speed processor with deep pipelines, branch prediction and complex memory hierarchies [2]. Moreover, the combined use of heterogeneous RISC, DSP and application-specific processors is often selected to deliver a given function on the most appropriate processor architecture class. Finally, these processors will be closely coupled with highly parallel hardware accelerators used for simpler, regular, fixed functions with high computational requirements.

This class of parallel, heterogeneous multi-processor SoC (MPSoC) platform needs to be programmed quickly and effectively, making efficient use of available resources. This is the key objective of an MP-SoC platform programming model and the associated platform mapping tools.

In the next sections, we survey parallel programming models used in large scale distributed systems. We examine the common requirements for SoC-scale embedded systems. We then describe the *DSOC* programming model and *MultiFlex* mapping tool developed at STMicroelectronics. This approach will be shown to provide many of the features of distributed systems, but with the performance and low-cost required in competitive SoC markets like 4G. Finally, we describe the application of the MultiFlex tools for the architecture exploration of a next-generation Nomadik^{TM} mobile multimedia platform developed at STMicroelectronics.

3. Complexity and Change

Designing and programming next generation wireless systems will become increasingly complex, due to heterogeneous multi-processor platforms, and the interactions with hardware accelerators. In addition, evolving standards and market dynamics requires solutions that can adapt quickly to changing requirements.

Complexity and change issues have been with us for some time in various guises, and partial solutions from related disciplines may apply. For example, in the traditional computing domain, complexity and change problems were addressed by distributed object models [3], as exemplified by approaches such as CORBA [4] and DCOM [5]. Refinements to these systems have introduced component concepts, as evidenced by CORBA Components [6], Sun's Enterprise Java Beans, and Microsoft's .NET, for example.When reflecting on the software industry's approach to dealing with complexity and change, it appears that a number of broad strategies are used, as summarized below.

Openness

In simple terms, openness means the system can be easily extended and modified. Strategies for achieving openness are as follows:

- Compose the application into components with well-defined functionality, and well-defined, well-documented interfaces.
- Use standards wherever possible (languages, development tools, etc.).
- Design the component so it can be executed in different contexts (e.g., by a component on another machine, on a different instruction set architecture, on another operating system, etc).

Heterogeneity

CORBA/DCOM and evolutions are built from the ground up to deal with heterogeneity. Microsoft .NET has perhaps the most aggressive strategy yet, with the common run-time library (CLR), which abstracts out most platform and language differences. Heterogeneity results from:

- **Different instruction set architectures (ISA):** ISA differences are a natural result of market forces and evolution in computer technology. (*Note: in the SoC environment, additional factors tend to increase ISA heterogeneity. These include the widespread practice of tuning an ISA for a particular application domain, in order to increase performance and/or lower power consumption, area, etc.*)
- **Different programming languages:** In the SoC environment, this is not as bad as the general S/W context, due to the overwhelming use of C. However, in the future, we believe the limitations of C will result in a move to a more heterogeneous mix of languages.
- **Different operating systems:** Again, in the traditional software domain, OS differences are a natural result of market forces and OS evolution. *In the SoC environment, OS diversity increases due to new OS variants tuned to embedded devices (Symbian, embedded Linux, Nucleus, Windows CE, VxWorks etc.), plus innumerable "home grown" custom operating systems.*

Transparency

Another key strategy for dealing with change and complexity involves the notion of "Transparency" [3]. This concept was first articulated in [7], and is a key aspect of the International Standard on Open Distributed Processing (ODP) [8]. The different aspects of transparency form levels with dependencies, as illustrated in Figure 1.

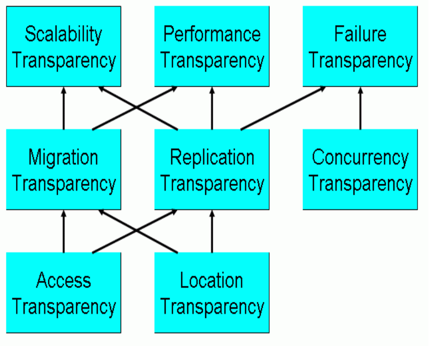

Figure 1: Transparency Dependencies

The different aspects of transparency are briefly summarized here as follows.

Access Transparency

Access transparency means that the interface to a component is the same, regardless of target architecture or language that it is implemented in.

Location Transparency

Location transparency means components can be identified in service requests without knowing where the component is located.

Location transparency is key to building scalable distributed systems. If the location of the component is hard-coded into the application, the component cannot be easily moved to adapt to changing system loads, changing functional requirements, etc.

Migration Transparency

It may often be necessary to change the resource that is executing a particular component, in order to meet real time execution, power requirements, changing functionality requirements, etc. Migration transparency refers to the fact that components can be moved to new resources without the component or interacting components knowing or taking any special action. Obviously, migration transparency is not possible if a source component needs to use a new interface if a target component is moved (e.g., the target object does not provide access transparency). Similarly, migration is not possible if a source component has a hard coded location of the target component (e.g., the target is not location transparent).

Replication Transparency

A service component may be replicated on a number of hosts, in order to meet performance requirements, real time constraints, etc. Replication transparency means the source component does not know or need to take any special action for target components that are replicated. Replication transparency implies the system has access transparency, and location transparency.

Concurrency Transparency

A distributed system may have many different components executing at the same time. Concurrency transparency means several source components may request a target component. The integrity of the service component should be preserved and application programmers need not see how concurrency is controlled.

Scalability Transparency

Scalability means the system can continue to operate as more resources are added, more services are added, and more concurrent requests are active. Scalability transparency is enabled by other transparency aspects such as replication transparency and migration transparency.

Performance Transparency

Performance Transparency means it is transparent to users and programmers how performance is actually achieved. The system could employ a variety of mechanisms related to the above transparency aspects in order to achieve performance transparency. For example, it could migrate components, perform load balancing over resources, power up new resources, etc. Components should be written without knowing or caring about the mechanisms used to achieve performance. Therefore, performance transparency usually implies migration transparency and replication transparency.

Failure Transparency

Failure transparency implies failures can be concealed from users and component designers. This usually implies replication transparency, so a component may fail, and the source components switch over to replicated target components without any special action on part of the source components. Also, failure transparency usually implies a transaction can be restarted in some way, which in turn implies concurrency transparency.

4. Programming Models in the Context of SoC

The challenge of managing complexity in previous generations of technology advances in VLSI systems has usually been managed by creating insulation layers between various levels of abstractions, as illustrated in Figure 2. In the eighties, the standard cell library was defined to insulate the logic designer from the physical design layer and process-specific details. In the nineties, a register-transfer level (RTL) abstraction level was defined, along with a new generation of synthesis tools. This has become the de facto abstraction for hardware logic circuits. The counterpart of RTL for embedded software is the instruction-set architecture (ISA) specification. This allows to program a complex processor architecture from a well-defined interface.

The current generation of SoC's requires a higher-level insulation layer which we refer to as a platform programming model. This should allow system developers to define high-level application descriptions without in-depth knowledge of the underlying heterogeneous HW/SW platform on which it will be mapped.

This need to formalize, simplify, and optimize the hardware/software interface to support higher levels of abstraction is of course not a recent discovery. Much pioneering work in this area has already been done in academia, e.g. TIMA laboratories [9], in the electronics industry, e.g. Philips Research Labs [10], and by commercial ESL design technology companies, e.g. the interface synthesis technology of CoWare^{TM} [11].

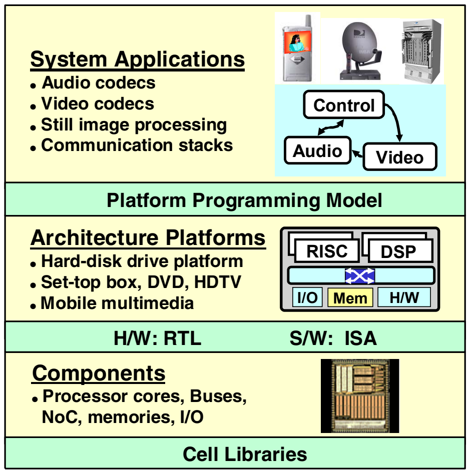

Figure 2 – Platform Programming Model

However, much work remains to be done before these concepts become standardized and mainstream, and broadly applied across the industry. One difficulty is that the motivation for this sort of approach comes from the more abstract "system-level". The advantages are not as clear when taking a locally optimal perspective. This more confined viewpoint discounts the longer term benefits such as the ability to handle change and adapt to new contexts. Similarly, engineers on the software side of the divide must learn to restrict the many layers of abstractions, and move towards simple and efficient interfaces that can unify hardware and software components.

5. The DSOC Platform Model

Our particular approach to the problem makes explicit reference and acknowledgment of the many valuable lessons and concepts from early work in distributed object systems, such as CORBA [4] and DCOM [5].

On the other hand, we have consciously chosen extremely demanding target applications, e.g., 10 Gb/s IPv4 packet forwarding, 2.5 Gb/s traffic management [12], and MPEG4 video encoding [13], in order to focus on the boundary layer between hardware and software.

The synthesis of CORBA concepts with the extreme performance of hardware accelerated applications has resulted in what we call a "platform programming model" called DSOC. We briefly review the DSOC concepts here, additional details can be found in [12].

DSOC stands for "Distributed System Object Components". As in CORBA and DCOM, an application is composed of a set of objects with well-defined interfaces. We will use the application and platform example of Figure 3 to illustrate the basic concepts. The top of the figure depicts a simplified application, consisting of control, video and audio application objects. These objects need to be mapped to the underlying MP-SOC platform. In most cases, the user limits the scope of the mapping. For example, in this case the control functions are mapped in many-to-one fashion to a general-purpose processor, running a standard O/S. The imaging objects are mapped in many-to-many fashion onto an array of DSP's (not running an O/S). The *Video* object is mapped to a hardware component.

In CORBA and DCOM, object interfaces are expressed in an Interface Definition Language (IDL). In our case, the IDL is called *SIDL* (System Interface Definition Language), and is a C++ subset. We have chosen a C-based syntax, in order to simplify the learning curve for ESW developers.

Each object exposes one or more interfaces that may be used by clients. In addition, each object may require access to external services, which satisfy some particular interface. In line with the transparency requirements of Section 3, the location of clients and servers associated with an object is of no concern to the implementation of the object itself.

We have developed an interface compiler, which is able to process interface declarations, and generate/interface with various component communication hardware and software. Various communication patterns are possible, ranging from in-process direct connections (e.g., as between objects *Control1* and *Control2* in Figure 3), inter-space connections, connections in a data-flow pipeline structure (e.g., as between objects *Imaging1, Imaging2* and *Imaging3* in Figure 3), or client/server connection patterns.

However, our focus is on extreme efficiency, and extremely light-weight implementations. For example, to support the client/server communication pattern, we have developed a hardware implementation of the object request broker. This broker matches client objects requesting services, with server objects that can provide the requested service.

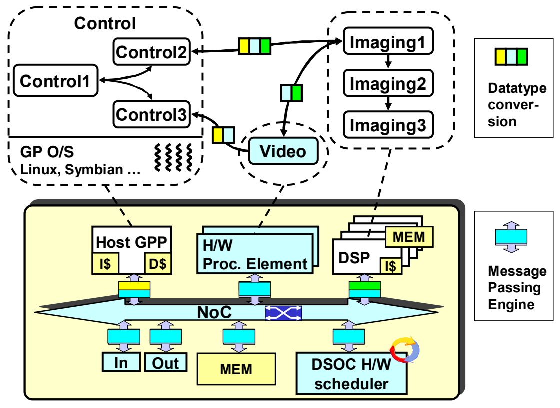

Figure 3 - DSOC Concepts

Also note that an object implementation may be in hardware. For example, in Figure 3, object *Imaging1* may communicate through the broker with object *Video*, which is implemented in hardware. Objects *Imaging1* and *Imaging2* may communicate through a direct binding. From the perspective of object *Imaging1*, it does not know (or care) that object *Imaging2* is local and is implemented in software, and object *Video* is remote, implemented in hardware, and communication is mediated through a broker. This enables some of the transparency aspects as described in Section 3. This transparency allows easy reconfiguration of the system, to enable different hardware/software mappings for changing requirements (different applications, quality of service, etc.).

Figure 3 also shows a "message passing engine". This is a hardware implementation of the "marshaling" operation found in distributed object systems. This operation takes all of the arguments for a remote object call, and converts them into a neutral portable format, and sends over the network on chip. The message passing wrapper on the receiving object performs the inverse operation. Due to the hardware acceleration, these remote object calls are very efficient. For a call with a few parameters, a remote object call can be done in a handful of instructions – roughly the same overhead as a normal object call in a language such as C++. Our first implementation of the object request broker hardware is around 0.05 mm^2, and each message passing wrapper is around 0.40 mm^2 (both figures are for implementation in a 90nm CMOS process).

We believe the DSOC platform programming model is an example of how to resolve the efficiency/abstraction conflict, as presented in Section 4. A high efficiency DSOC platform provides the abstract, high-level, object-oriented communication mechanisms directly in the platform architecture, without the many levels of indirections and inefficiencies. This enables direct and efficient communication between components implemented in HW/SW, HW/HW, and SW/SW. Implementations may be changed between HW and SW, remapped on to different resources, and/or load balanced across a set of resources, with a minimum impact on the design and implementation of the actual components. This enables the transparency, openness and heterogeneity strategies as outlined in Section 3, and provides an efficient platform architecture that is better equipped to deal with the emerging change & complexity requirements.

6. Application to the Nomadik Mobile Multi-Media Platform

The STMicroelectronics Nomadik platform is an interesting early example of the trends we have discussed in this paper. As shown in Figure 4, our target *exploration* platform is composed of

- A general-purpose processor running a standard operating system,
- A 3D subsystem with embedded DSP
- A video subsystem with an embedded DSP
- An audio system with dual embedded DSPs.
- System interconnect
- Various Memories

This platform is targeted at mobile applications, so high efficiency operation is essential. Therefore, a great deal of functionality in the video and 3D subsystems is implemented in hardware.

On the other hand, the platform must be very flexible. Media pipelines are becoming very complex, with multiple instances executing in parallel, each with multiple processing stages, and various quality of service/power tradeoffs, etc. The device operation is dynamic, in that new pipelines must be started, stopped, and reconfigured, as the device is running.

In addition, previous experience has show that new requirements usually emerge after each generation is brought to market. This typically requires rebalancing of the computation, and perhaps sharing resources in ways that were not originally intended.

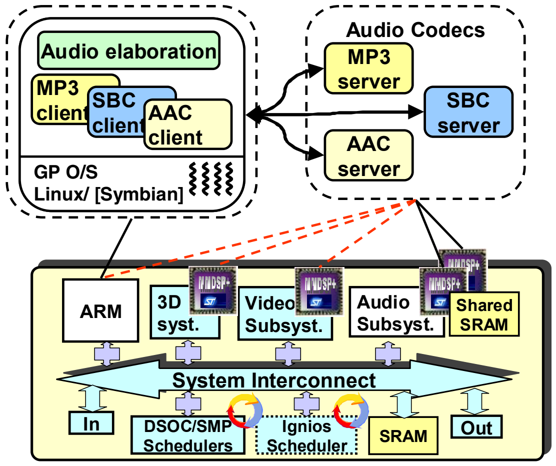

Figure 4 - Nomadik Mobile Multi-Media Platform

These change and flexibility requirements have motivated the exploration of a DSOC-style approach. To illustrate this, we implemented an audio processing example, with three different types of audio decoders (AAC, MP3 and SBC), distributed over two DSPs. The source and final target components for each flow are on the ARM processor, which ran Linux in our experiment (the Symbian DSOC flow is currently under development). The source components read the encoded data from an input file, and the target components store the decoded PCM data to the output file.

In this example, a static binding of components to resources may not result in optimal processing throughput. For example, consider the static binding with AAC and SBC on one DSP, and MP3 on the other. If a burst of SBC and AAC frames occur, with a momentary pause in MP3 frames, one DSP will be fully loaded, and the other idle. Therefore, we enabled the DSOC load balancing, which allows both DSPs to service requests from all three audio decoder application components.

Table 1 - DSP Utilization

Function	Utilization (%)
AAC decode	50
MP3 decode	40
Idle	8
SBC decode	1
Data Transfer	0.9
Pipeline elaboration	0.09
DSOC overheads	0.01

Table 1 shows the DSP utilization for the various tasks. Note the small overheads associated with DSOC support. Also, the DSP is idle for 8% of the time. This was in fact due to ARM file I/O overheads (3 file sources and 3 file sinks). We simulated this configuration with a much higher clock frequency for the ARM processor, and this overhead went to zero, achieving a 1.97X speedup on the dual DSP platform over a single DSP platform.

Table 2gives the DSOC code size, or "footprint", for the various DSOC layers. This is the static code size, not the number of instructions to execute the particular function. Note that ARM code size is not as sensitive as the DSP, as the ARM has techniques (virtual memory, paging, etc) to deal with code size. On the MMDSP, we see the average footprint for each client interface method is 38 instructions, and the server size is 46 instructions. The methods have two or three arguments each, some of which are frame buffers. With the DSPs, we are using the non-blocking DSOC interface, which requires around 5 times more code than the most efficient binding (available for hardware multi-threaded processors). Note that this overhead is per-class; multiple objects can exist for each class type, and only pay the code size overhead once.

Table 2 - DSOC Code Footprint

	ARM Processor	DSP Processor
App-specific user-mode DSOC Client Code	43 instructions per method, avg	38 instructions per method, avg
App-specific user-mode DSOC ServerCode	65 instructions per method, avg	46 instructions per method, avg
DSOC libs	2991 instructions	466 instructions
System call overhead	963 instructions	N/A

Even though these footprints are quite tiny, compared with other distributed object or component systems, it is still a cause for concern. Work is currently underway to extend the compilers and other parts of the DSP tool chain with component support. With this approach, components are dynamically composed and configured on the ARM processor, and deployed on the DSP processors, with essentially 0 run time support required on the DSP.

These experiments with the Nomadik platform provide a good example of how we believe the resolution of the abstraction and efficiency issues will be handled in the future: We see embedded software moving towards higher object-oriented and component abstractions, while simultaneously removing software layers and overheads, achieving increasing efficiencies.

7. Conclusion

We have described the common subset of requirements for distributed software systems, as illustrated by approaches like CORBA and DCOM, and those of SoC-scale embedded systems, namely: openness, heterogeneity and multiple forms of transparency.

We have demonstrated that it is possible to apply these concepts to the emerging class of complex, parallel SoC's which will be used in 4G wireless systems. These platforms will include multiple heterogeneous embedded processors interacting with hardware co-processors and I/O devices. An implementation of this approach is embodied in STMicroelectronics' DSOC (Distributed System Object Component) programming model and the associated MultiFlex platform mapping tool. The use of this programming model for an architecture exploration of ST's Nomadik^{TM} mobile multimedia platform was described.

References

[1] P. Magarshack, P.G. Paulin, "System-on-Chip Beyond the Nanometer Wall", *Proc. of 40th Design Automation Conference (DAC)*, Anaheim, June 2003.

[2] Kunio Uchiyama, Pradip Bose, "Energy-efficient Design", *IEEE Micro Magazine*, Vol. 25, No. 5, Sept-Oct. 2005, pp. 6-9.

[3] Wolfgang Emmerich, "Engineering Distributed Objects", *John Wiley & Sons, Ltd.,* 2003

[4] Object Management Group, www.omg.org.

[5] Distributed Component Object Model (DCOM), http://www.microsoft.com/com/tech/DCOM.asp

[6] CORBA Components Specification 3.0. Technical Report formal/02-06-66, Object Management Group, June 2002.

[7] ANASA (1989). The Advanced Network Systems Architecture (ANSA). Reference manual, Architecture Project Management, Castle Hill, Cambridge UK.

[8] ISO 10746-1 (1998). Information Technology - Open Distributed Processing – Reference Model: Overview. International Standards Organization.

[9] A. A. Jerraya, F. R. Wagner, W.O. Cesario, "Hardware/Software Interface Design for SoC", in Embedded Systems Handbook, CRC Press, 2006.

[10] P. van der Wolf, E. de Kock, T. Hendriksson, W. Kruijtzer, G. Essink, "Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach", in Embedded Systems Handbook, CRC Press, 2006

[11] See CoWare web site: http://www.coware.com/solutions/hwsw.php

[12] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, "A Multi-Processor SoC Platform and Tools for Communications Applications", in Embedded Systems Handbook, CRC Press, 2006.

[13] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, G. Beltrame, G. Nicolescu, "Parallel Programming Models for a Multi-Processor SoC Platform Applied to Networking and Multimedia", *submitted to IEEE Transactions on VLSI Journal*, 2005.