Platform Design Approach for Re-configurable Network Appliances

R.Cmar, R.Paško, J-Y.Mignolet, G.Vanmeerbeeck, P.Schaumont and S.Vernalde

IMEC, Leuven, Belgium

Abstract

The presented platform-based object-oriented modeling concept for system design allowed us to create a networked hardware re-configurable camera in a 25 man-month schedule with concurrent development of application and target FPGA platform. The developed TCP/IP layer achieves throughput of 2Mb/s/MHz and the complete application logic consumes 700 mW at 20MHz.

Introduction

Internet has become a driving force for embedded electronics. Software based embedded systems often do not offer the best solution in terms of power and speed. Our alternative design approach with emphasis on distributed and parallel hardware solutions uses a C++ based high-level system design methodology. It encourages implementation of dedicated network protocols embedded in an application. A standalone, all hardware networked camera with network re-configurability was designed to demonstrate this concept. The implementation is FPGA-based and offers good performance/power/flexibility tradeoff.

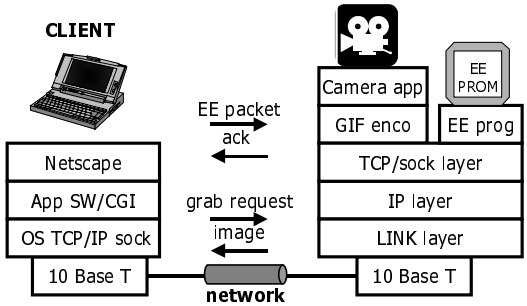

Figure 1: Application structure

The networked camera design consists of several layers (Figure 1). The network connectivity is provided by the LINK, IP and TCP socket layers [1]. Data acquisition from the attached camera is triggered from the network by an 'image request' packet. The camera responds with image data, which is gif-encoded and sent back to the client. The FPGA is re-configurable at run-time through the EEPROM programming layer. Reconfiguration data can be downloaded, altering the application architecture and/or functionality. The client side is software-based and connects with the camera over the internet, for instance with a netscape session.

Motivational case

The overall architecture is shown in Figure 2 with some design options illustrated in the IP layer. First the handshaked versus fifo-buffered byte transfer (*Com*) is shown in the transmission path, then the distributed versus shared memory organization (*Mem*) in the data reception path. Similarly, the memory organization option appears in the TCP socket layer.

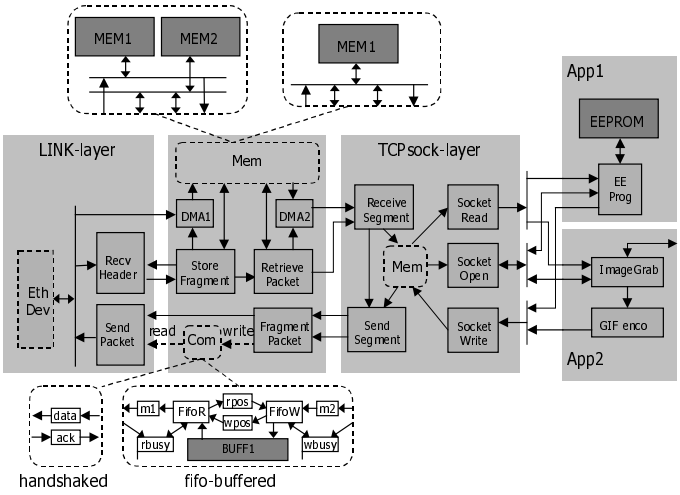

Figure 2: NetCam architecture

Such options are often being fixed at early design phases and then they influence the rest of the design development. The simple example is the *Com* component. At a strategic position in the design it might strongly affect the design performance. The fifo-buffered solution with an optimal depth can introduce desirable pipelining effect. It is not trivial for a designer to predict the best positioning and depth in case of multiple instantiations. Hard choices are often made with memory organization (*Mem*). Once chosen, the memory bandwidth is determined, so are limits on an achievable processing performance. Knowing these limits, designers often stick to an ad-hoc solution in deciding on system parallelism.

In our design flow we keep the options open and then validate them at the system level as design parameters. To support this concept, we applied design methods such as layered abstraction, object oriented approach with advanced reuse mechanisms. We begin with a highly distributed processing model described behaviorally, which through system options can be scaled down and mapped onto a specific platform.

For *Mem* components we employed a dynamic memory management (DMM) technique, which can have various implementation schemes, ranging from generic one to application specific. In our application, the reception path of the IP layer is not critical as it passes only configuration data and image requests, so we have chosen a minimal memory scenario. On the contrary in the TCP layer we used the distributed scenario with two memory units implemented with different DMM schemes. Related to the *Com* component, we found

for instance, that its positioning between *FragmentPacket* and *SendPacket* increased the average performance by 24%.

New design modeling

The all hardware implementation of a traditionally software design [2] was enabled through OCAPI-xl, a C++ design library. It raises the abstraction level of design by introducing high-level behavioral objects. The concept is based on a timed multi-threaded C++ system description (similar to [3]) with fully automated implementation backend. Behavioral objects are processes (threads), instructions, loops, conditions, messages, variables, semaphores, etc., forming a HW/SW unified model, open towards hardware or software implementation. In this particular case the FPGA target [4] determined the hardware path. An intelligent HDL code generator understands the object interactions and derives appropriate architecture elements. Examples: processes are transformed into control/datapath (FSMD) architectures; messages into blocking-read protocols with register; semaphores into separate processes with request/grant I/O interface; shared variables into bus accessed registers.

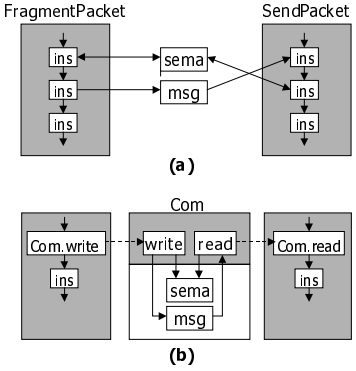

Figure 3 : Object oriented communication case

In OCAPI-xl, a process is described with a customized instruction sequence (ins). Processes communicate through specialized instructions operating on messages (msg), semaphores (sema), and shared variables (svar). An access to external resources (memory, camera) is modeled with a foreign language interface (FLI) mechanism, providing simulation model and synthesised into a system port. Figure 2 shows *Com* component as a communication means between *FragmentPacket* (P1) and *SendPacket* (P2) processes. Figure 3a illustrates the detailed implementation with one message and one semaphore object. P1 waits on the semaphore before sending data while P2 releases the semaphore after reading out data. This simple communication results in 1-item deep queue implementation.
We can build the *Com* component for reuse (Figure 3b) through virtualization of the interface, similar to [5]. The implementation details of the communication become hidden and only the *read/write* accessing methods are exposed to the designer of P1 and P2. During the code generation the accessing methods get expanded into P1 and P2 processes and automatically adapt I/O interfaces on both sides. Both *Com* implementations from Figure 2 are designed this way, i.e. the handshaked and fifo-buffered. The latter one expands into two processes FifoR and FifoW operating on a dual port memory (BUFF1) being synchronized with two shared variables (rpos, wpos). The I/O behavior is expressed with two semaphores (rbusy, wbusy) and messages (m1, m2).

FPGA platform based approach

The complete system implementation involves an FPGA core with an off-the-shelf ethernet controller, two external SRAMs, EEPROM memories and a camera connector on a single PCB board (Figure 4).

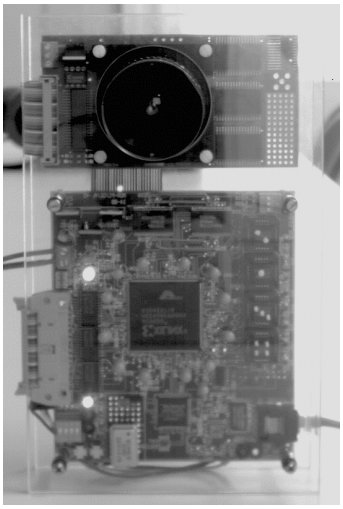

Figure 4: PCB design

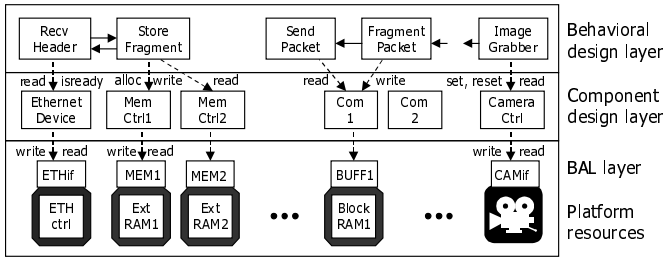

Figure 5: Layered platform design

In our platform design we used a three-layer hierarchical design model (Figure 5). The board abstraction library (BAL) is created as a collection of objects associated with available resources on the board (including internal FPGA memories, i.e. BlockRAMs). For instance, the *ETHif* interface object defines the accessing protocol to the ethernet controller and encapsulates it into *read/write* methods. The BAL layer forms the basis on which complex architecture elements can be created like *EthernetDevice*. The component layer is made up out of these architecture elements, which express actions like bus transfer through e.g. *write()*. This practice allows design for reuse and further abstraction. The behavioral design layer finally defines application specific processing in a highly abstracted description. As all the three design layers are uncoupled with well-defined interface definition it allows for concurrent design development.

Design concept for reuse

The object oriented (OO) modeling, provided by OCAPI-xl, supports design abstraction, flexibility and scalability.

Abstraction: Encapsulation is applied on objects of both BAL and component layers (Figure 5). E.g. *ImageGrabber* is provided with high-level functions like *set*, *reset* and *read* of the underlying *CameraCtrl* component, while *CamIF* object hides specific cycle-accurate protocol through the camera interface before *CameraCtrl*. Using virtual methods supports hierarchical design and makes an application description portable to different platforms.

Flexibility: Depending on the nature of a data transfer several alternatives might be considered for the *Com* component. The component interchangeability is enabled through virtualization of read/write accessing methods, regardless of what I/O interface or protocol is used.

Scalability: We described the IP layer processing with a distributed access using two memoy units. Each memory unit is accessed through a dedicated memory object (*MemCtrl)*, which provides memory managing virtual functions. The processes like *DMA1*, *StoreFragment*, etc, access *MemCtrl1* or *MemCtrl2* or both with such virtual functions. The distributed (parallel) memory organization can be switched to a shared (sequential) one, by directing all function calls to e.g. *MemCtrl1* only.

In the next session we will exemplify these principles on few implementation cases in the TCP/IP layer.

Implementation cases

A. Communication expansion

The choice on the *Com* implementation is done at the system level. The two examples of the handshaked and the buffered communication between *FragmentPacket* and *SendPacket* are depicted in Figure 6. It is shown that not only the implementation is different, but the interface (ports) are automatically adapted accordingly.

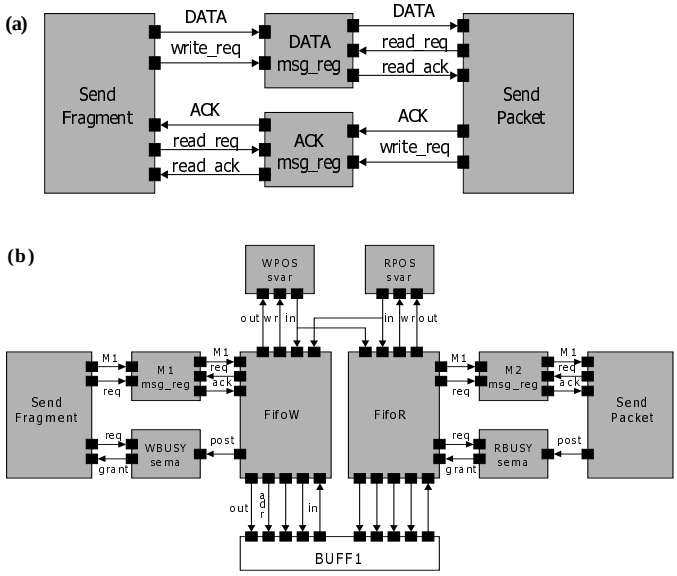

Figure 6: (a) Handshaked and (b) Fifo-buffered communication expansion

B. Dynamic memory management (DMM)

The TCP/IP by nature requires highly dynamic data manipulations and encourages parallelization, which however, might be limited by memory organization.

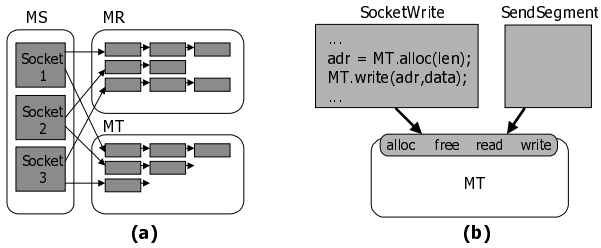

Figure 7: (a) Application specific memory partitioning into multiple memory banks (b) DMM accessing to the memory bank

Closer look into data processing of the TCP layer (Figure 7a) reveals we can logically partition memory into at least three memory banks (MS, MR, MT). MS holds a pool of socket structures, where the socket structure represents a connection. MR maintains a queue of data segments flowing from the IP layer to the application layer, while MT contains data segments for the opposite flow. Different processes access different memory banks. E.g., *SocketRead* accesses MR, while *SocketWrite* accesses MT, which justifies parallelization.

In our approach each memory bank is linked to a memory object with certain DMM mechanism. The memory object provides a virtual interface represented with *alloc*, *free*, *read* and *write* methods for TCP (or IP) processes. Applying DMM allows elegant and address-independent manipulation with data segments and socket structures. Possible conflicts in case of parallel access are resolved with semaphores, which are parts of the accessing methods.

For implementation of the DMM as a generic solution we used *free-list table allocation*. It requires a memory map, which holds the information of free memory blocks inside of the data memory in a list (Figure 8a). This DMM is designed with three OCAPI-xl processes (*Arbiter*, *New*, *Delete*) and attached memory resource(s) (Figure 8b). In our implementation we placed both the memory map and data memory into a single RAM.

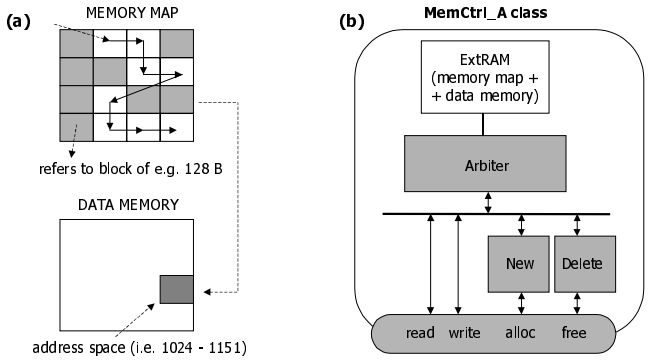

Figure 8: (a) DMM scheme with free list allocation table (b) its implementation with three processes and one RAM resource

C. Multiple DMM implementations

We can think of different DMM schemes and then apply the most suitable, perhaps application specific one. For example, the socket structure in our case is fixed to 89 bytes. Let us assume we will support only limited number of connections, e.g. 5. Instead of using external RAM this scheme can fit into the BlockRAM (of 512B) applying a specific DMM, where table is degraded into register with 5 bits (Figure 9a).
Another example applies for the IP layer if we decide not to support fragmented packets. The memory object degrades into a simple buffer (BlockRAM of 2048B for one IP packet) with a flag register to indicate buffer occupation (Figure 9b).

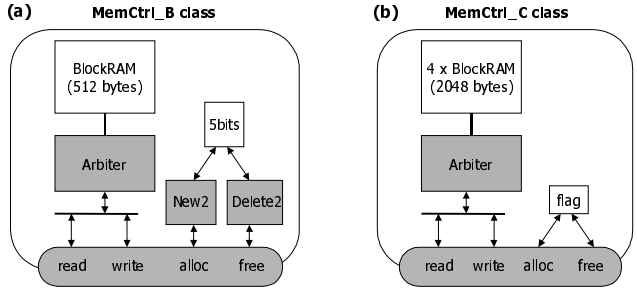

Figure 9: Application specific DMM implementation with (a) 5 bit allocation register (b) 1bit flag register

Note that the behavioral layer of the TCP/IP is amenable to such changes, while the implementation becomes less complex and performance increases.

D. Parametrizable adaptation to a platform

Our TCP/IP is described as a highly distributed processing with distributed memory organization (two memory banks in the IP layer and three memory banks in the TCP layer) in a storage resource uncommitted way. Targeting a specific platform we need to map our behavior onto limited number of available storage elements (ExtRAMs, BlockRAMs). The memory mapping is done at the system level, without affecting the behavioral level, with the following options available:

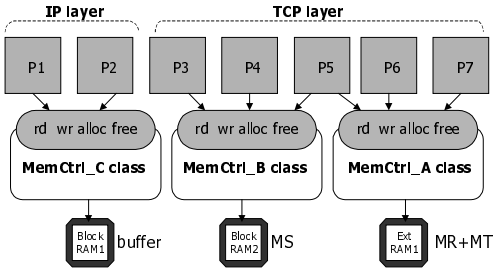

Figure 10: Chosen Mem mappings for TCP and IP

1) Switching between different DMM schemes poses different requirements on storage elements. E.g. we implemented the MS bank with the *MemCtrl_B* class DMM scheme, which maps onto BlockRAM instead of using expensive external memory.
2) Multiple memory banks can be merged into the shared configuration under common DMM scheme. This reduces potential parallelism while decreasing demand on resources. E.g. we linked the MR and MT banks into the shared *MemCtrl_A* DMM scheme, as the reception traffic across MR was negligible for this application.

The taken approach allows parametric scaling down from the distributed scenario into the shared one. The configuration decision for our application is shown in Figure 10. Other configurations can be easily derived and immediately evaluated for performance by cycle-accurate simulation.

Results

The application was described with 38 threads at the behavioral layer with multiple instantiation of 5 reuse component elements. The hierarchy and abstraction of the design flow allowed for concurrent design development with 25 man-month schedule starting from the algorithmic phase including the PCB design. The automatically obtained VHDL was compiled and resulted in 79 kG, containing TCP/IP, re-configurable engine, camera interface and gif-encoder. The clock frequency is 31 MHz for 0.5u/3.3V MIETEC technology. The mapping onto Virtex family architecture resulted in 5322 CLBs.
The setup with reduced TCP layer with clock frequency of 20 MHz resulted in 700mW power consumption (2W including camera) at a 3.3V (5V for camera) power supply. The simulated throughput of TCP/IP reaches 2 Mb/s/MHz. The real throughput of 4 Mb/s limit was caused by available camera module interface. Currently we are building the new platform with a fast camera interface (over 80Mb/s), which will enable us to validate high bitrates.

Conclusion

We presented an object-oriented concept for platform design. We took the approach of the highly distributed processing with the layered design abstraction. This allows a parametric mapping onto a big range of platforms with a different set of requirements, e.g. memory resources. We focused on the networked camera implementation with the flexible TCP/IP core. We targeted the FPGA platform, which encourages re-configurability. Starting from the generic TCP/IP we can derive different architectures, which might be applied dynamically per application case. This high-level approach supports design for reuse and drastically reduces development time. At the same time it keeps advantages of hardware design, resulting in a good performance and lower power consumption.

References

[1] R. Stevens: TCP/IP Illustrated: Volume 1 and 3, Addison-Wesley, 1994 and 1996.
[2] Axis communications: Axis network camera, www.axis.com/ products/camera_servers.
[3] SystemC: www.systemc.org
[4] Xilinx: Virtex family: datasheet reference, www.xilinx.com/ products/virtex.htm
[5] D.Gajski et al: SpecC, Kluwer Academic Publisher, 2000