Mapping the Physical Layer of Radio Standards to Multiprocessor Architectures

Cyprian Grassmann*; Mathias Richter **; Mirko Sauermann*
*Infineon Technologies AG COM PS CE; **Siemens CT PP 2

Abstract

We are concerned with the software implementation of baseband processing for the physical layer of radio standards ("Software Defined Radio - SDR"). Given the constraints for mobile terminals with respect to power consumption, chip area and performance, non-standard architectures without compiler support are the targets a SDR implementation has to face. For this domain we present a way to safely move from a functional model to the assembly level in order to come to a tested multithreaded optimized implementation in manageable time.
We carried out this program for the standards WLAN IEEE 802.11b and 3GPP WCDMA exploiting various levels of parallelism: thread level parallelism ("MIMD"), data level parallelism ("SIMD") and instruction level parallelism ("VLIW"). We came up with a software implementation running in real time on Infineon's programmable Multiple SIMD Core (MuSIC) processor.

1. Introduction

Mobile terminals are facing very tight constraints with respect to chip area, power consumption and performance. This is especially true if the radio standard has to be implemented in software to cope with the ever raising demand for flexibility and the handling of multiple standards on the same platform.
We take it for granted that the key to achieving high performance at low clock rates (low power consumption) lies in the exploitation of parallelism. The physical layer of radio standards exposes inherent parallelism at various levels:

- the sequence of processing steps for transmission and reception is repeated for many data elements; repetitions can be overlapped in time
- components of transmitters and receivers like FIR filters expose data parallelism in themselves
- instructions for data transfers, arithmetic and communication can be issued in parallel (LIW)
- special instructions finally address the bit-level parallelism

An architecture correspondingly offering support for thread-, data-, instruction-, and bit-level parallelism at the same time will be all but standard. Capabilities and performance potential are great, but the price to be paid is a serious complication of the software development process. Notably the asynchronous nature of thread level parallelism is not easy to deal with. It is difficult to be exploited optimally and to have guaranteed maximal execution time. Above all, it is not easily tested for functional correctness [1].
It would be desirable to start from an abstract functional description and system modeling of the respective radio standard and proceed through some mapping process to the final implementation - ideally supported by a compiler, with an automated extraction of the parallelism.
Practically the development of a compiler would take a very long time and only should be considered if a final agreement on a feasible architecture for SDR has been found.
The main difficulty is the wealth of possible mappings from a function to its implementation. An optimal mapping is defined in terms of a minimal number of arithmetical instructions and memory transfers and a minimal amount of processor communication - all at the same time. The mapping is highly sensitive to the underlying hardware capabilities. Therefore it is necessary to actually (manually) carry out and profile the implementation to get a reliable estimate of its run time. Such crude measures as arithmetical complexity of a function can largely fail to give a true picture.
On the other hand, we do need a high level model: When a specific carefully optimized implementation reveals a performance bottleneck, we can not simply enhance the hardware. In fact the SDR promise is to keep the same hardware for many (new) radio standards. Rather, algorithms (or parameters of it like filter lengths or data precision) used in the application need to be reworked so that they allow for a more efficient mapping. To save development time, new algorithms are tested against requirements of a radio standard at the highest possible level before they are again mapped to the hardware. This whole process is iterated.
We have followed the approach presented in this paper for the WLAN standard IEEE 802.11b and for

3GPP WCDMA-FDD on the application side and for Infineon's multiprocessor "MuSIC" on the architecture side.
In the next section we outline the MuSIC architecture and its implications on software. The following section describes the considerations that drive our partitioning and mapping decisions. Section 4 gives details on how we actually implement and test at different levels and how we assure a smooth and safe transition from higher to lower levels. The final section summarizes the benefits achieved by our approach.

2. Architecture and Programming Model

We only consider baseband processing and omit all analog components of a communication system.
An analysis carried out at Infineon Technologies [2] resulted in proposing the MuSIC processor as a versatile baseband processor meeting the performance, power, and area restrictions mentioned above.

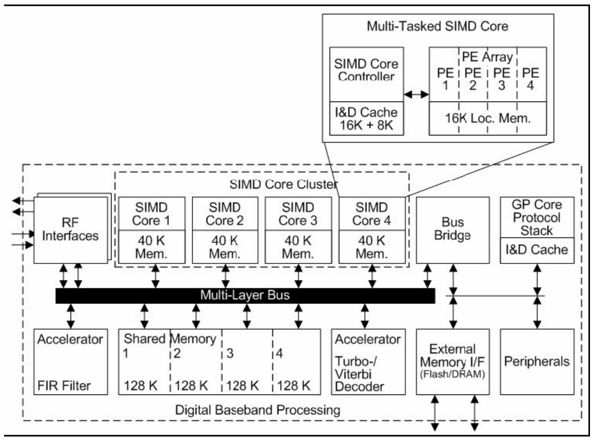


Figure 1: Baseband platform

This processor chiefly consists of a cluster of four single-instruction multiple-data (SIMD) DSP cores (Figure 1). Each SIMD core contains four processing elements (PE) and operates with a clock frequency of 300 MHz. To relax timing requirements for the memory and to resolve pipeline hazards each core runs four threads which are switched by a fixed time multiplexing mechanism. It can thus be assumed that in a whole we make use of 16 threads running at 75 MHz each.
Long instruction words (LIW) of the PE array show memory, arithmetic, and communication components. The SIMD core controller in fact is a 32 bit general purpose processor (GP). The GP communicates with the other units via instruction and data FIFOs.
The cluster of SIMD cores is accompanied by dedicated configurable accelerators for channel encoding and decoding as well as filtering operations. In addition, there is an ARM processor for the execution of the protocol stacks.
For a detailed discussion of the MuSIC processor, see [3], [4], and [5].
The programming model for this architecture is that of multithreaded programming in C. Wrapped into functions called by threads, the purely data parallel parts (associated with SIMD cores) are programmed in a data parallel language extension of C.
To support multithreaded programming, we have developed our own light weight operating system ("ILTOS"). It provides the means to create and synchronize threads, to asynchronously send and query messages between them and to allocate and free shared memory.
Functions to be executed on a SIMD core are written in DPCE, the Data Parallel C Extension language [6], a superset of the C language. DPCE offers parallel data types and operations on them. We have developed a compiler which takes a DPCE source and produces synchronized C code for GP core and DMA transfers between shared and local memories (to be translated further by a C compiler for the GP) as well as PE assembly. This compiler is not yet optimizing, though. To achieve best performance we use inline assembly for the PE array and explicit DMA configuration. What remains is chiefly the C language with some intrinsic functions for PE and DMA control plus an assembly source code library for the PE. Implementations can be done completely without PE by writing pure C programs. These will then run on the GP core alone. This feature is of importance for testing assembly implementations, as shown later.
At Infineon, we also have developed a virtual prototype of the entire MuSIC platform based on SystemC. The virtual prototype is a cycle- and bit-accurate software-based simulator. It contains models of all processors, accelerators, busses, memories, and peripherals which will be available in the real hardware. Therefore the same software can be run on the virtual prototype as on the real hardware.

3. Modeling and Mapping Radio Standards

We start with a dataflow diagram of the application. A dataflow diagram or DFD consists of processes, denoted by rectangles, data flow, denoted by arches and external entities. The direction of the arch represents the flow of data from producer to consumer. The semantic is that a process can fire when all its inputs are available [7]. In addition to other modeling approaches with DFDs we require that all processes are free of side effects (referentially transparent). Due to this restriction we will further refer to processes as functions. To convert a process with state into a function, we make internal states externally visible.

As example take a FIR filter process, which continuously filters incoming samples. Such a filter process usually has internal state, which stores the history of the incoming signal. In order to get rid of this state, the filter gets an additional output, which is fed back as input. The feed back signal needs to be delayed, such that it is used along with the next input block. For this reason we introduce a delay element.

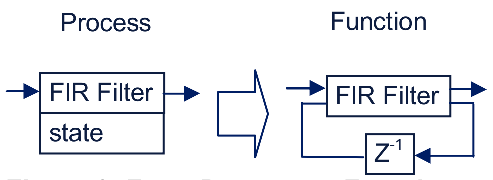


Figure 2: From Process to Function

The restriction to functions is actually no real restriction of the computational model, as a functional model is equally expressive. On the contrary, by making states externally visible we model the flow of state data explicitly, which was hidden in the black box of the processes in the original DFD. This important change allows us to get a complete dependence relation between functions.

Let's take the following DFD as an example for further discussion

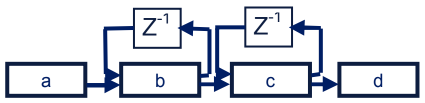


Figure 3: Example

The next step towards a multithreaded program is the analysis of the dependence relation between functions, since only dependences restrict the schedule and thus the available parallelism within the application. Since we are talking about streaming applications, which work on blocks of continuous data, we map each chain of producer-consumer functions in the DFD into a two dimensional linear space. One dimension is, like the original DFD, the chain of different functions. The other dimension represents the function instance, which is associated with the incoming data block. Note that delay elements now become superfluous as the producer instance is now directly connected with the according consumer instance.

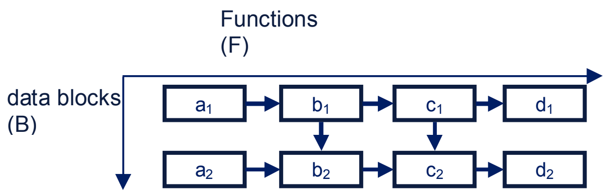


Figure 4: Expanded DFD

Each arrow represents an element of the dependence relation on the set of function instances. The dependence relation can be expressed as set of vectors:

$$D = \{ \vec{d} = \vec{F}_i - \vec{F}_j : function\ \vec{F}_i\ depends\ on\ \vec{F}_j \}$$

The goal of the mapping step is to find two functions, commonly referred to as Time Mapping (Schedule) and Space Mapping, which map the function instances in a particular order to a particular thread or processor. Both mappings can be summarized as a function M from our two dimensional linear space Functions x Block(Instances) into the two dimensional linear space Time x Processors.

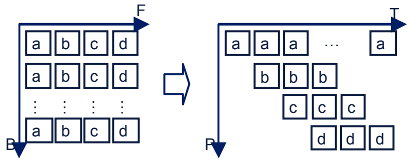
$$M : F \times B \to T \times P$$

Apparently such a mapping is only valid if it respects the dependences. That is a function instance is only allowed to be evaluated if all functions which are smaller (with respect to the order induced by D) are already evaluated. More formally:

$P_T(M(\vec{d})) > 0\, \forall \vec{d} \in D$ where P_T is the projection to the T dimension

In our implementation we particularly look at linear mappings, for linear mappings M can be represented as matrices. Of particular interest are mappings which minimize execution time and thus throughput. For the given example there are two such mappings.

The first one is characterized by the following transformation:



$$M\begin{pmatrix} F \\ B \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} F \\ B \end{pmatrix}$$

Figure 5: Affine Mapping I

Such a mapping is well suited, if the processors are fixed function ASICs, because there are only different instances of the same function mapped to a particular processor.

Indeed, in our 802.11b project we used this kind of mapping for functions which are destined to run on hardware accelerators.

The second mapping is characterized by the following transformation:

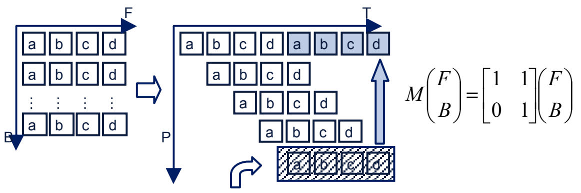

F
T
B
P
a b c d
$M\begin{pmatrix}F\\B\end{pmatrix}=\begin{bmatrix}1&1\\0&1\end{bmatrix}\begin{pmatrix}F\\B\end{pmatrix}$

Figure 6: Affine Mapping II

It may seem that this mapping requires more processors. However processor 1 will get idle once it finished evaluating function d for the first time and can then proceed with a next evaluation. This is formalized by a second mapping, $P \mapsto P \bmod P_n$, where P_n is the number of physical processors. M then becomes piecewise linear.

The second mapping will often be preferable for programmable hardware because it can be better load balanced when, in contrast to our abstraction, functions do not have equal execution time. As an example assume the execution time of function b is three times as long as that of each function a, c, and d. In this case it is sufficient to use only 2 processors as any more would only increase the idle time.

In addition this mapping may reduce communication, since all data but “states” stay local on a processor.

A last important mapping parameter is the size of input blocks for the streaming application. For instance a filter function can process one or more input samples and produce a corresponding number of output samples at each call. It is plausible that the synchronization overhead of a multithreaded implementation decreases linearly with the size of the input blocks. One would like to choose as large blocks as possible. On the other hand the communication standard may restrict the latency from antenna to final bit detection. It is clear that latency increases with block size. The best solution here is to pick a block size as large as possible, which still fulfills the latency requirements. A good block size can only be found by profiling the functions on target hardware.

Summarizing, we found that a mapping consists of two parameters. One is the mapping function M, which determines parallelism and communication costs and thus throughput. The other is the block size, which determines latency.

4. Implementation of the Iterative Design Flow

We advocate a three level approach, tacitly assuming a hierarchy of parallelism, comprising a high level functional system model for algorithmic testing, a multi threaded reference model (in C language) running on a workstation to cope with the problems of asynchrony and nondeterminism and an assembly implementation running on the target hardware or a simulator of it for exploitation of data level, instruction word and bit-level parallelism. All levels are kept throughout the iterative software design process.

At the high level system model signal processing algorithms exposing sufficient parallelism are selected, arithmetical (fixed/floating point, division, square root, ...) and data precision (bit width) restrictions of the hardware are considered and the model is tested against the functional requirements of a standard (e.g. maximal error rates of a receiver).

The analysis of the model, as it is described in section 3 then leads to a partitioning of the application into multiple threads - the multithreaded reference model. Again this multithreaded reference program is tested for functional equivalence with the functional system model.

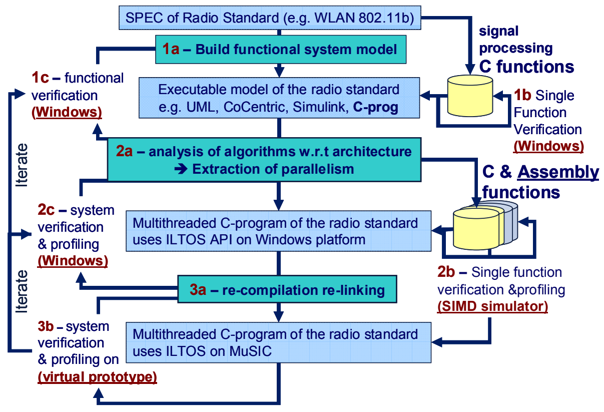

SPEC of Radio Standard (e.g. WLAN 802.11b)
signal processing C functions
1a – Build functional system model
1c – functional verification (Windows)
Executable model of the radio standard e.g. UML, CoCentric, Simulink, C-prog
1b Single Function Verification (Windows)
Iterate
2a – analysis of algorithms w.r.t architecture → Extraction of parallelism
C & Assembly functions
2c – system verification & profiling (Windows)
Multithreaded C-program of the radio standard uses ILTOS API on Windows platform
2b – Single function verification &profiling (SIMD simulator)
Iterate
3a – re-compilation re-linking
3b – system verification & profiling on (virtual prototype)
Multithreaded C-program of the radio standard uses ILTOS on MuSIC

Figure 7: Software design flow

Based on experience and profiling of the multithreaded reference model, critical functions of the implementation are picked to be implemented as an optimized assembly implementation by exploiting the given data level and instruction level parallelism within a thread function. The functional correctness is tested against the reference function of the model. The final implementation, including the assembly functions is then profiled on the hardware or a simulator of it for testing against the real time requirements imposed by the radio standard under investigation. In this case the profiling can be limited to the critical use cases to save simulation time.

It is important to point out that we could create a hardware independent functional model without caring about algorithmic details, like using only “signal processing exposing parallelism” or hardware preferences for certain arithmetical operations, but this usually leads to a decoupling of the functional model from the rest of the development and substantially reduces the benefit of that model. We rather also iterate over the functional model, as it has to be a concrete algorithm that has to be proven feasible to

cope with the requirements of a standard and that has to be mapped to hardware.
Our highest level of abstraction is a dataflow diagram. C implementations (e.g. S functions in case of Simulink models) of DFD nodes have to be furnished to capture algorithmic details and the block size parameter. Implementation decisions are tentative when they rely on performance measurements not yet available and must eventually be reconsidered. The block size parameter is an example for this, as already pointed out in section 3. Nevertheless we will already profile at this level to get an idea of how well C functions would meet performance requirements and where optimized assembly implementations will be inevitable.
The next step consists in mapping to a multithreaded reference program, our second level of abstraction. We use exactly the same function implementations as for the DFD to assure functional equivalence.
Where it seems appropriate we finally go down to the lowest level, namely assembly implementations. Assembly usually is frowned upon as a programming model. From our experience we argue, though, that assembly programming is tedious, but manageable in terms of development time (as compared to the development of dedicated hardware for every new communication standard) and *safe* in terms of provable functional correctness – especially if it is limited to critical sections identified beforehand, by profiling the application.
Compared to this, ensuring correctness of a multithreaded program is a much more challenging task because of the effects of nondeterminism. It is absolutely hopeless to test a multithreaded program on a non-standard hardware (simulator) when only restricted debugging support is available. This is true all the more when simulation speed is a concern.
From these experiences we have drawn the following conclusions:

1. We develop and test a multithreaded program on a standard platform with good tool support. We have chosen the Windows operating system. It must be ensured that an application tested on Windows will also run correctly on the target hardware.
2. We develop assembly code only for single functions called from within a thread. We test assembly functions for correctness against their C reference counterpart, using the virtual prototype simulator of the hardware.
3. We profile only tested applications to find out about timing correctness.

When the profiling results are satisfactory, we are done. Otherwise we have to rethink about assembly implementation, multithreading, or even the functional model, and go again through the list of corresponding lower level steps in the design flow. In the rest of this section we describe how we actually carry out the steps 1 to 3 described above.

4.1 Migration from Windows to target system

The key to a quick and safe transition from Windows to our target system lies in the ILTOS operating system. The underlying simple idea is to restrict ILTOS to a minimal and portable API offering support for multithreaded programming. As already mentioned in section 2 the API consists of functions for thread administration, memory management, communication, and synchronization.
All API functions can easily be mapped (by a preprocessor) to corresponding Windows operating system calls. A multithreaded program can then be compiled for Windows as well as for MuSIC (as described in section 2, this program will run on the GP core only and won't make use of the PE array). Testing and debugging will be done under Windows, using the Visual C++ development system and additional tools like Intel thread checker. Correctness of the transition to MuSIC depends only on the correctness of ILTOS and the MuSIC C compiler. But these are developed and tested once and for all and have already evolved into a steady and reliable state.
We cannot stress enough that testing the correctness of a multithreaded program is the most difficult part of the software development process. It is a great advantage that this part can be carried out on a well established system with all possible tool support and has not to be done for a special target system.

4.2 Assembly implementations

Extracting data parallelism contained in functional units is the next step to take, where this is necessary to meet real time requirements.
Assembly code is wrapped into a function in form of inline assembly together with C code for the general purpose (GP) core. These functions are compiled for MuSIC and run on the virtual prototype (VP) simulator. The VP provides several tools for analyzing software. GP cores can be debugged by using a graphical debugger. Further, internal state, instructions, and data transfers of all modules (GP, PE array, DMA transfers) can be traced and logged. We rely on the virtual prototype behaving exactly like the hardware. A proof of this is not part of the software development process.
For testing purposes, functions are called for various sets of input data and corresponding outputs (which can be complete sets of states) are saved. The produced outputs are then compared with those produced by corresponding C reference function.
We mention explicitly that testing assembly implementations is alleviated by the fact that it has not to be

done for the full application but only for parts of it having a clear interface in terms of input and output data.
When this is accomplished, we can deliberately link either pure C functions or functions containing inline assembly together with the multithreaded skeleton of our application. Only the first version will run on both platforms (Windows and MuSIC), but both will deliver the same result for MuSIC.

4.3 Profiling

The final task is testing for real time behaviour of the application by profiling it. This is done again using the VP.
The VP can track the execution status of threads on the general purpose processors by monitoring calls of ILTOS synchronization functions. It is not necessary to change the application software for this. The results can be displayed graphically. An example of an 802.11b implementation running on the MuSIC DSP cluster is shown in Figure 8. The traces show single threads running, blocking, and synchronizing.

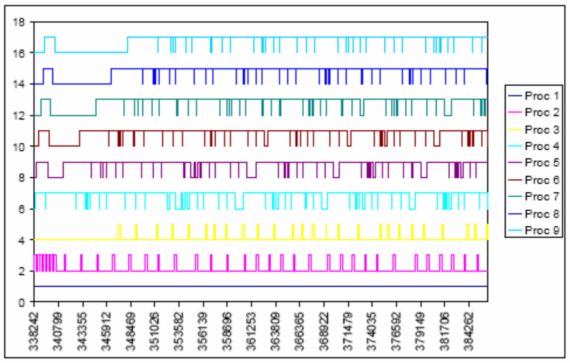


Figure 8: Traces of parallel WLAN 802.11b implementation.

For a more detailed analysis, the virtual prototype can gather statistical data on the utilization of the shared modules like busses and memories. From the trace files written we can also get a cycle accurate picture of the relative timing behaviour of PE, GP and DMA transfers.
Having these analysis tools we come quickly to a very accurate analysis of the performance of an application and can spot out performance bottlenecks and balance the usage of the system resources.

5. Conclusion

The presented development methodology leads to a reduction and an essential acceleration of the unavoidable design iterations for the implementation of radio standards on non-standard architectures. Especially the well specified method to identify the right block size and to find suitable mappings, as described in section 3, leads to a reduction of the design search space.
Remaining design iterations are accelerated by early testing on standard platforms (with good tool support) and by the reuse of implementations and test beds from the functional system model down to the assembly implementation level. This is possible by providing a portable OS API, by incorporating C-reference functions already within the functional system model, and by keeping the model within the iteration loop.
By consequent usage of this model based approach in conjunction with concrete implementations of the functions involved, interface contracts can be worked out at an early stage of the design process. Hence the approach also substantially supports the distribution of the implementation onto multiple developers and lays the foundation for automated generation of the error prune multithreaded control code. We experienced this in the case of WCDMA, where three separate development teams worked on functional modelling and on C and assembly implementations.

6. References

[1] E.A. Lee, *The Problem with Threads*, IEEE Computer, 39(5):33-42, May 2006
[2] H.-M. Blüthgen, C. Sauer, M. Gries, W. Raab, D. Langen, A. Schackow, M. Loew, U. Hachmann, N. Bruels, U. Ramacher, *Finding the Optimum Partitioning for Multi-Standard Radio Systems*, SDR'05 technical conference, Orange County, California, 2005.
[3] W. Raab, H.-M. Blüthgen, U. Ramacher, *A Low-Power Memory Hierarchy for a Fully Programmable Baseband Processor*, WMPI 2004.
[4] H.-M. Blüthgen, C. Grassmann, W. Raab, U. Ramacher, J. Hausner, *A Programmable Baseband Platform For Software-Defined Radio*, SDR'04, Phoenix Arizona.
[5] H.-M. Blüthgen, C. Grassmann, U. Ramacher, *A Software Programmable Multiple-Standard Radio Platform,* IST Mobile Summit, Dresden, 2005.
[6] DPCE 1.6 Technical Reference, www.crescentbaysoftware.com/dpce/index.html
[7] W.P.Stevens, G.J.Myers, L.L Constantine, *Structured Design*, IBM Systems J., 1974, 13 (2) pp. 115—139
[8] C. Grassmann, A. Troya, M. Sauermann, M. Richter, U. Ramacher, *Mapping waveforms to mobile processor architectures,* SDR'05 technical conference, Orange County, California, 2005.
[9] M.Sauermann, ILTOS API Reference v0.7 Documentation, Infineon internal report, 2006