SOFTWARE PARTITIONING AND HARDWARE ARCHITECTURE FOR MODULAR SDR SYSTEMS

Arnd-Ragnar Rhiemeier, Friedrich K. Jondral
Universitaet Karlsruhe (TH), Karlsruhe, Germany
{rhiemeier, jondral}@int.uni-karlsruhe.de

ABSTRACT

In this paper we investigate the relationship between logical and physical structures of a Modular Software Defined Radio (Mod-SDR) system, based on multi-processor hardware. The logical interdependence of software modules and functions in the signal processing chain is captured in a directed graph where both input/output data and intermediate results flow along directed arcs and where nodes represent software modules or entire communication functions. While the need for data exchange is obvious on a logical level, its mapping to physical resources is not. Computation and data communication require synchronization such that the SDR terminal can provide best performance to its user. Hence, our object of interest is partitioning and scheduling for software defined physical layers, which is a core component of any SDR operating system firmware. We briefly review a partitioning approach [1] which we have developed for application in the SDR domain. Subsequent scheduling is treated in greater detail. Taking the graph of an IMT-2000 W-CDMA 64kbps uplink as an example we explore hardware architecture options when using an application-specific co-processor, and we discuss the achievable system speedup. Finally, we arrive at a conclusion regarding subsystem pipelining and dynamic power dissipation in CMOS hardware implementations.

1. INTRODUCTION TO MODULAR SDR

Benefits and challenges of Software Defined Radio have been discussed extensively in the past [2][3][4]. The focus of most work is either on one specific hardware part of the signal processing chain of a transceiver, or on algorithm design for the digital baseband. However, coordination of the interplay of communication functions, to be executed on available hardware, has not been systematically covered hitherto. Therefore, our research interest is in structures and algorithms for optimum design of software defined physical (PHY) layers in mobile terminals. This includes design guidelines for both hardware and software. From our point of view the term “software” means digital signal processing modules which are executed under the direct control of real-time operating system (RTOS) firmware. The execution of such modules generally entails allocation of hardware resources such as processors, memories, buses etc. An early document of the SDR Forum [5] roughly sketches the process of resource allocation (see Fig. 1), however, not specifying techniques required to achieve such a goal.

1.1. Modularity

Regarding 3G mobile communication systems even rough estimates produce enormous numbers of FLOPS for the processing power required to realize even a most simple PHY layer in software. Given the technological challenges to provide such extreme processing power we believe that it may be commercialized soonest with multi-processor hardware. Whether later such a multi-processor system is realized as a System-on-Chip, or as an FPGA with immersed CPU cores, or as a pure multi-DSP board, is not for us to say. At the moment, it is just important that our approach to Mod-SDR design scales smoothly with all these variants and with technological advancement.

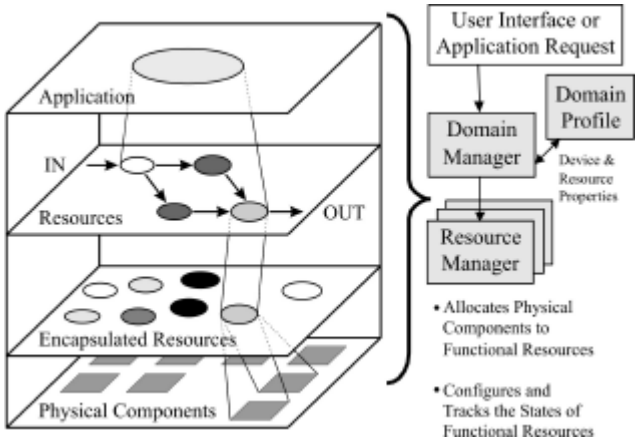

Fig. 1: Layered Resource Allocation, Source: SDR Forum [5]

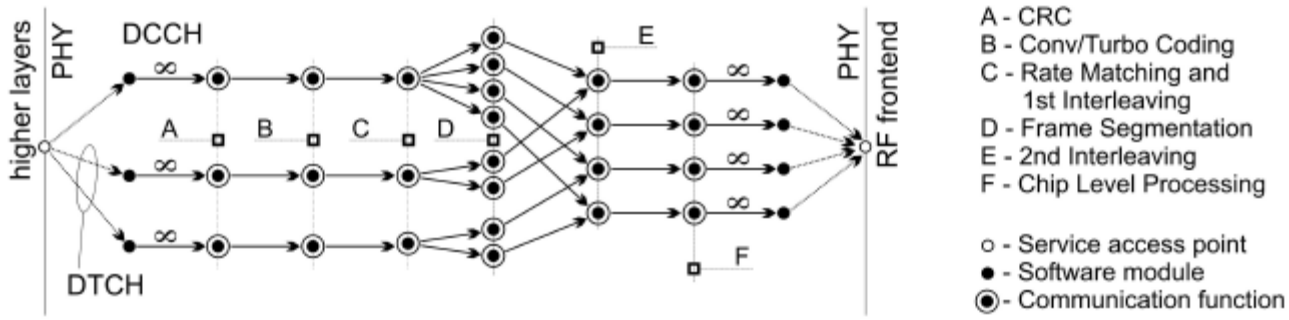

Fig. 2: Directed graph of an IMT-2000 W-CDMA transmitter, 64kbps uplink mode, abstracted from [6]

Our model of software radio is modular in two ways: First of all, in an open multi-layer SDR platform software is certainly not one monolithic block of machine-executable code, but rather a hierarchical library of communication functions, subfunctions and – at the lowest level – modules, which are basically indivisible sequences of machine code. Second, the underlying SDR hardware may also be modular in a way that it can be composed of general-purpose digital signal processors as well as Application Specific Standard Product (ASSP) components, according to customers' needs. This view of Mod-SDR may be interesting to equipment manufacturers who are naturally skeptical SDR techniques. In particular, the availability of a unified flexible operating system could not only support short time to market, but also amortization of software development cost over several product lines.

1.2. Related Work

In previous work [7] we have focused on non-preemptive scheduling of a multi-processor, where inter-processor communication was assumed to be infinite-bandwidth and collision-free. In the present paper, we render the model more realistic by considering finite data exchange bit rate and collision arbitration on a dual bus hardware system. The proposed methods are related to static partitioning and scheduling of software modules which are characterized by strong logical interdependencies through data flow and precedence constraints. Hence, dynamic scheduling policies such as EDF and LLA [8], or combinations and variants thereof, have no helpful meaning in this context because these methods act on strictly independent processes. In addition to that, many of these algorithms assume a preemptive operating system.

In [7] we argue in detail why SDR is essentially an embedded system design problem, where the real-time requirements are strictly imposed by the radio interface of a requested communication mode. It is imperative that the PHY layer RTOS is also a decision system. The firmware accepts or rejects a mode request, based on its successful or unsuccessful partitioning and scheduling among available terminal hardware. Once a mode is accepted, the sequence of functions, subfunctions, and – eventually – modules is fixed, and so is the demand for processing power. In this context, we see no eventuality of statistical demand for processing power [9]. Nevertheless, it is not known to the partitioning and scheduling algorithm a priori which kind of communication mode a user might want to request, which hardware configuration may have been set up, or how much battery power that user is willing to sacrifice. In other words, our approach must be capable of dealing with diverse physical and logical structures, and operate modular systems under difficult boundary conditions.

2. GRAPHS

In general, processors, memory, buses (or any other interconnect fabric) form the entirety of what we call the physical structures of an SDR. Logical structures, in contrast, are captured in a directed graph where both input/output data and intermediate computation results flow along directed arcs and where nodes represent software modules (solid dots), or entire communication functions (circled dots). Figure 2 shows the example which we refer throughout this paper: The transmitter of an IMT-2000 W-CDMA terminal operating in a 64 kbps dedicated traffic channel (DTCH) mode in the uplink. The given logical structure is an abstract transcription of [6]. In contrast to [7], both modules and functions are uniquely numbered on the top-level hierarchy of the graphical representation.

The central point in our approach is to perceive these logical structures from a processor point of view. Whatever the software design approach behind a module or function, it is scheduled for execution at a certain instant of time, it consumes input data, it causes a processing runtime to pass on a processor, and it produces output data which usually constitutes some intermediate result to be consumed by successor modules. Whether the module contains parameterized code [10][11] or standard-specific DSP code, or even hardware-specific instructions, does not matter to us. All that counts is its main behavioral attribute, namely processing runtime.

2.1. Resource-Runtime Model

In the present paper we employ the following stochastic linear resource-runtime model:

$$p_m = a \cdot c \cdot r_m \quad (1)$$

where p_m is the processing runtime of module m and r_m is m's resource demand of local memory for its output data. The constant α is a processor-specific runtime, and c is a unitless random factor modeling the vast variety of software implementations in an SDR environment [12]. Throughout this paper, c is drawn from a random process with windowed Gaussian probability density, where μ_c=1.0, σ=0.25, and the rectangular window spans the interval [0.5; 1.5] of c values. That is to say: actual implementations of modules may randomly differ by ±50% from the strictly linear resource runtime relation at μ_c=1.0. Whether runtimes are indeed caused by running software modules or result from using dedicated hardware does in no way alter our modeling of Mod-SDR. Therefore, it may be seen as a complement to other SDR design paradigms, for example, PaC-SDR [13], or as a basic model for a more general theory of flexibly assembled modular systems. As such it may be of interest not only to SDR software programmers, but also to equipment manufacturers and system designers.

3. BASIC HARDWARE SYSTEM FEATURES

Figure 3 shows our assumed hardware architecture which includes two general-purpose processors (P_1 and P_2), an application-specific co-processor (CO) and distributed memory (M1, M2 and M). Inter-processor communication is asynchronous over the Shared Memory which is used to store intermediate results. Processors are actively involved in bus transfers, i.e. a processor transferring data to/from the Shared Memory cannot execute communication functions in the meantime.

The system is equipped with two separate data buses (solid horizontal arrows), where bus access is exclusive. In principle, all processors be connected (vertical arrows, both solid and outlined) to both buses. However, regarding the hardware implementation effort it would be beneficial to provide only a single bus connection per processor (solid arrows).

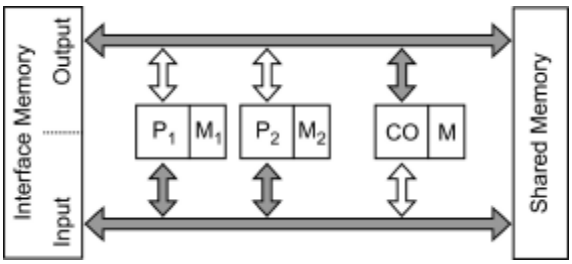

Fig. 3: Physical structures of the Mod-SDR system

Data enter and leave the system via the Interface Memory. Input data (bits of some protocol data unit, for example) arrive at the PHY layer service access point (SAP) for processing during the real-time frame ΔT. Likewise, output data (I/Q signal samples) have to be passed to the analog RF frontend every ΔT. In order to ensure that interacting hardware parts can unconditionally write their exchange data to some memory location during all ΔT, the interaction at the system interface is assumed to follow a pipelining concept: While the hardware system is active on one part of the interface memory, subsequent data arriving at the interface during ΔT are written to the shadow memory part of the interface. At real-time frame boundaries pointers to the active and the shadow memory are merely swapped, so processing can continue immediately on both sides of the interface.

In contrast to former memory organization [12], where shadow memory had been part of the Shared Memory, we save some unnecessary i/o bus transfers and make inter-system interaction much simpler. Hence, in the following we can safely deal with the PHY layer, which is now a well-defined embedded system to the wireless terminal.

4. PARTITIONING

The proposed approach for partitioning software modules among processors in a Mod-SDR is based on a fundamental algorithm by Kernighan and Lin (KL), which had originally been developed to distribute electronic components among printed circuit boards while minimizing the cost of inter-board connections [1]. Formulated in an abstract way: Devide a node set $\mathbb{S}$ of 2n nodes into two disjoint subsets $\mathbb{A}$ and $\mathbb{B}$ of size n each, called partitions. Component connection cost is collected in a symmetric cost matrix $\mathbf{C}=[c_{ij}]$, $1 \le i,j \le n$, where $c_{ij} = c_{ji} \in \mathbb{R}$ are link cost between nodes i and j. Link cost can be either external or internal, depending on whether the link is between nodes of the same set or reaches into the other set, respectively. The basic idea of the KL Algorithm is to gradually reduce external cost by exchanging node pairs between $\mathbb{A}$ and $\mathbb{B}$ until there is no more positive gain from their set exchange procedure.

Link cost in our Mod-SDR can be identified with a processing runtime p_{ij}, which is a *property of an edge* $<i,j>$ and *not* a node property like p_m. In particular, p_{ij} be the time that one processor spends communicating to the Shared Memory instead of computing useful results. Not only the source processor has to write, but also the destination processor has to read data from the Shared Memory. Hence link cost c_{ij} is twice the time p_{ij} of one processor. A detailed description of our adaptation of the KL Algorithm to Mod-SDR can be found in [12].

Should a partitioning cut separate two connected nodes, link cost c_{ij} is resolved into two basic bus transfer modules with runtime p_{ij} which is tied to the bus speed rather than to the processor speed. Furthermore, the runtime p_{ij} is *not* subject to stochastic variation, but deterministically depends on the amount of data to be transferred over the bus. Accordingly, we introduce the relative bus speed β indicating how much faster some data block is transferred over the bus as compared to its being processed by either of the general-purpose processors at c=1.0. In the experimental section the relative bus speed β re-appears as a major hardware parameter relevant for system performance.

5. SCHEDULING

The adapted KL Algorithm allows partitioning of modules among two identical processors P_1 and P_2 connected by a dual bus system. However, the sequencing of modules is as yet undetermined at this stage. Only now scheduling comes into play.

Our approach makes use of Hu's labeling idea [14]. Nodes are labeled by their so-called Hu Level which is the maximum accumulated runtime distance to a graph's target node (the SAP of the analog RF frontend, see Fig. 2). In contrast to Hu's original algorithm, however, scheduling decisions are taken on a per-processor basis because all nodes have already been assigned to a processor during the partitioning stage. The box to the right provides the pseudo-code of our scheduling algorithm which honors the prioritization of bus transfers, the existence of a dual bus system and the pipelining of signal processing over several radio frames. All of these aspects have been recognized [12] as being essential for meeting the exact design conditions of our partitioning approach.

6. SYSTEM PERFORMANCE

The design guidelines of this paper are intended to relate aspects of signal processing modules to the hardware architecture of Mod-SDR. In particular, we want to study the power-efficient use of an accelerated co-processor for an increase in system performance.

First of all, it is to be decided which nodes are good candidates for co-processor execution. It is easy to show by partial differentiation that the greatest decrease in aggregate processing runtime of the entire system results from accelerating nodes with greatest runtime, when co-processor acceleration is linear. Therefore, we assign the four chip level processing nodes (see Fig. 2) to the co-processor.

To be able to quantify the co-processor benefit in the experimental section we take into account the minimum runtime which is theoretically achievable by the hardware system: The aggregate runtime of all nodes, excluding all bus transfers and chip level processing, divided by the

```
 1 ActiveNodes
   = determine_successors_of_scheduled_nodes;
 2 ActiveNodes
   = sort(ActiveNodes | HuLevel/runtime);
 3 if (option_non_pipelined)
 4   ActiveNodes
     = remove_anticausal_nodes(ActiveNodes);
 5   if (is_empty(ActiveNodes))
 6     LOGICAL_WAIT_IDLE
         until not(is_empty(ActiveNodes))
 7   end
 8 end
 9 if (option_prioritize_bus_tx)
10   CandidateNodes = find(ActiveNodes | bus_tx);
11   if not(is_empty(CandidateNodes))
12     CandidateNodes
       = sort(CandidateNodes | HuLevel/runtime);
13     RunnableNode = CandidateNodes(top);
14     AlternativeNode
       = find(ActiveNodes | not(bus_tx))(top);
15   else
16     RunnableNode = ActiveNodes(top);
17     AlternativeNode = empty;
18   end
19 else
20   RunnableNode = ActiveNodes(top);
21   AlternativeNode = empty;
22 end
23 if (is_bus_tx(RunnableNode))
24   if not(conflict = check(bus_1))
25     NodeToSchedule = RunnableNode;
26     mark_busy(bus_1);
27   elseif exists(bus_2)
28     if not(conflict = check(bus_2)))
29       NodeToSchedule = RunnableNode;
30       mark_busy(bus_2);
31     else  %% both buses are busy
32       if is_empty(AlternativeNode)
33         PHYSICAL_WAIT_IDLE
           until earliest(not(conflict(bus_any)));
34       else
35         NodeToSchedule = AlternativeNode;
36       end
37     end
38   else  %% single bus is busy
39     if is_empty(AlternativeNode)
40       PHYSICAL_WAIT_IDLE
           until earliest(not(conflict(bus_1)));
41     else
42       NodeToSchedule = AlternativeNode;
43     end
44   end
45 else  %% no bus access required
46   NodeToSchedule = RunnableNode;
47 end
```

number of general-purpose processors. This minimum runtime still depends on the realization, but it is easy to compute during simulation. The ratio of this minimum runtime to the actual system runtime can be interpreted as the achievable fraction of the theoretical maximum speedup when using the co-processor.

7. ARCHITECTURE OPTIONS

Adding the co-processor means introducing a third processor to the system. However, our partitioning and scheduling algorithms have only been developed for a two-processor system. Generalizing the KL Algorithm is expected to be difficult, and much of the true co-processor benefit would probably depend on the scheduling policy which would have to be created for three processors running on two buses. Therefore, let us rather explore the theoretical bounds on achievable speedup, operating no more than the original two-processor system.

On one hand, a co-processor is only useful if it creates at least some non-negative speedup over the non-accelerated two-processor system. This case is readily covered by our model. On the other hand, no real system can perform better than a hypothetical system which employs an infinitely fast co-processor. The remainder of this hypothetical system is left with two identical processors P_1 and P_2, and data transfers over a finite-speed bus system, as before. Figures 5 and 6 show simulation results for the non-accelerated system and the hypothetical system, respectively. Both the achievable fraction of maximum speedup (dots, value range [0;1]) and the observed bus activity (circles, values in percent) are plotted as a function of the relative bus speed β. Sample size is 400 per β.

Experiments with a single bus system (not shown graphically here) yield comparable results, where the speedup is merely shifted to the right by a factor of about two. We recognize that the achievable fraction of speedup approaches 1.0 as β increases, but even the fastest co-processor suffers from a major drawback: a large amount of intermediate results still needs to be transferred over the dual bus system at finite speed.

7.1. Pipelined Co-processor Operation

An architectural approach to mitigate the observed bus transfer problem is to divide the hardware into two subsystems: the co-processor plus one bus merge to become the pipelined co-processor subsystem, while the residual system includes P_1, P_2 and the second bus. Pipelining for these subsystems is realized in much the same way as for inter-system interaction (see section 3), but now the system-internal Shared Memory must provide space for both active and shadow memory.

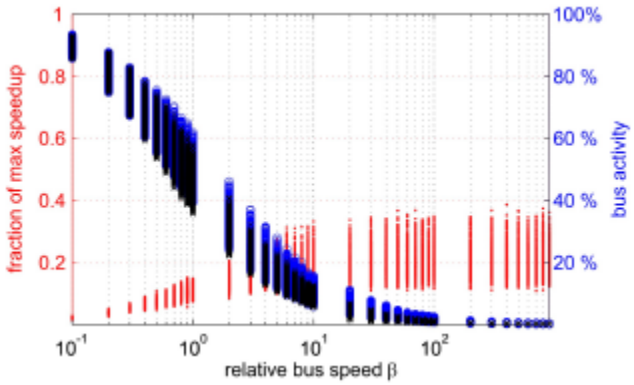

Fig. 5: Non-accelerated system, two buses

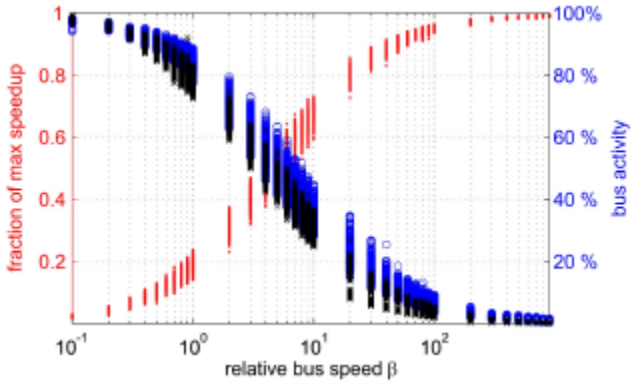

Fig. 6: Hypothetical system, two buses

The main advantage of this approach is that the residual subsystem must no longer the large amount of chip samples to the interface memory, but only the output data of 2nd interleaving (signal processing stage "E", see Fig. 2). A potential drawback consists in using a single bus only, because it violates the exact design conditions of our partitioning approach.

Figure 7 shows the fraction of achievable maximum speedup (dots), activity of the single bus (circles), and a realistic upper bound on speedup (triangles) as a function of relative bus speed β. In contrast to the theoretical maximum, the realistic upper bound takes into account all strictly serial bus transfers of the graph which is mapped to the residual subsystem. We observe that the achievable speedup fraction is close to the realistic upper bound and quickly approaches 1.0 for reasonable bus speeds. Formally, this advantage comes at the price of the additional delay of one real-time period ΔT, induced by the concept of subsystem pipelining. It is important to notice that the results of Fig. 7 only remain valid if the pipelined co-processor subsystem meets

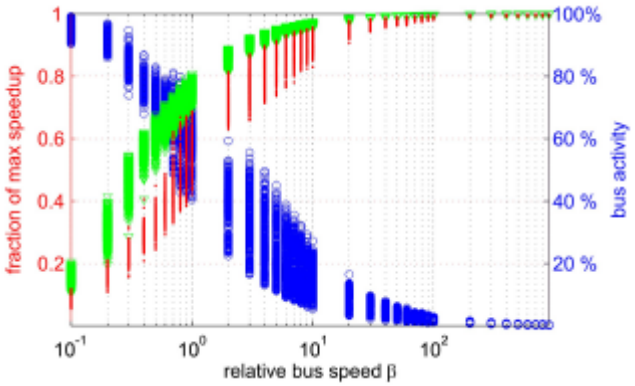

Fig. 7: Residual subsystem, one bus

the same (new) real-time deadline of ΔT, which is produced by the residual system. Therefore, we have to verify that the advantage of operating the residual subsystem at a low bus speed is not annihilated by any excessive clock speed in the pipelined co-processor subsystem.

8. DYNAMIC POWER DISSIPATION

The major drawback of high clock speed is dynamic power dissipation P_d [15]:

$$P_d \sim f_a \cdot C \cdot V^2 \qquad (2)$$

where f_a is the average switching frequency, C is the effective capacitance and V is the voltage level. We assume perfect clock gating during idle times, i.e. dynamic power is only dissipated during the true co-processor and bus activity. In principle, both the co-processor (C_0, f_0') and the bus (C_1, f_1') may be loaded with different capacitances and clocked at different speeds. A given deadline ΔT can be reached using any out of a continuum of clock ratios f_0' / f_1'. Hence it would be of advantage for Mod-SDR operation to know the optimum clock ratio, if such one exists.

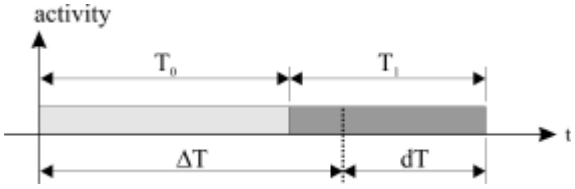

Fig. 8: Aggregate runtimes of co-processor (T_0) and bus (T_1)

Let us assume that the sum of aggregate runtime of the co-processor (T_0) and the bus (T_1) need to be reduced by an amount dT to meet the real-time period ΔT. Figure 8 visualizes the situation.

Let us further assume that the constant fraction r, $0 \le r \le 1$, of the required dT is covered by an increase in bus speed, and the rest of (1-r)·dT is covered by the co-processor. Then the frequency increase factor k_1 over a regular bus operating at relative speed β can be expressed as:

$$k_1 = \frac{T_1}{T_1 - r \cdot dT} = \left[1 - r \cdot \frac{g}{1-j}\right]^{-1} \qquad (3)$$

where γ is the target runtime ratio and φ is the co-processor portion of the sum of aggregate runtimes:

$$g = \frac{dT}{T_0 + T_1} \quad \text{and} \quad j = \frac{T_0}{T_0 + T_1} \qquad (4)$$

Similarly, frequency increase factor k_0 for the accelerated co-processor over one running at the specific runtime α is:

$$k_0 = \frac{T_0}{T_0 - (1-r) \cdot dT} = \left[1 - (1-r) \cdot \frac{g}{j}\right]^{-1} \qquad (5)$$

The dynamic power increase factor k_P for the subsystem is:

$$k_P = \frac{f_{a,0}' C_0 + f_{a,1}' C_1}{f_{a,0} C_0 + f_{a,1} C_1} \qquad (6)$$

where the average frequencies in denominator are $f_{a,0} = \varphi \cdot f_0$ and $f_{a,1} = (1-\varphi) \cdot f_1$, and in the numerator:

$$f_{a,0}' = f_0 k_0 \cdot \frac{T_0 - (1-r) \cdot dT}{T_0 + T_1 - dT} \qquad (7)$$

$$f_{a,1}' = f_1 k_1 \cdot \frac{T_1 - r \cdot dT}{T_0 + T_1 - dT} \qquad (8)$$

Expressing the timing relations of (7) and (8) in terms of (4) and substituting k_0 and k_1 by (3) and (5), the power increase factor k_P eventually turns out to be:

$$k_P = [1 - g]^{-1} \qquad (9)$$

We recognize that the increase in dynamic power dissipation is no function of φ, or of the fraction r, or of any effective capacitances C_0 and C_1, but of γ only. The actual γ value of a Mod-SDR realization depends on the real-time period ΔT produced by our partitioning and scheduling for the residual subsystem.

8.1. Simulation Results

Figure 9 shows the power factor as a function of relative bus speed β of the residual subsystem. Since at low β it mainly suffers from the bus bottleneck, the pipelined co-processor subsystem performs sufficiently well with moderate power factors. As bus speed grows beyond β =10 the spread among power factors becomes relatively stable, and the majority of values falls into the range $5 \le k_p \le 25$.

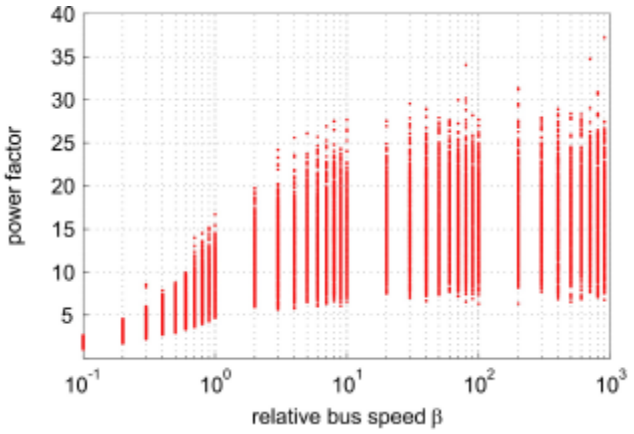

Fig. 9: Power factor for the pipelined co-processor subsystem

Finally, we need to compare these results with the power demand of the non-pipelined system, which is based on the *same* hardware. Therefore, let us set our common design goal to achieving 90% of the theoretical maximum speedup.

On one hand, when operating the Mod-SDR in a non-pipelined way, relative bus speed must be pushed to around β=50 (see Fig. 6) to achieve the design goal. Both buses are active about 10% of the real-time period ΔT. On the other hand, when using pipelined subsystems, β=10 already achieves the design goal, while the activity of the residual system's bus is about 20%. Under the assumption that two distinct buses are loaded with twice the effective capacitance of a single bus it follows that dynamic power dissipation of is by a factor of 5 lower for the pipelining approach.

To complete the power budget we also need to discuss P_1, P_2 and CO. Given our partitioning approach, P_1 and P_2 are almost non-stop busy, regardless of how the Mod-SDR is operated, so they equally add to both sides of the budget. Now, regarding CO, we have to consider its activity in the both the non-pipelined and the pipelined case. Our results indicate a finite demand of 5 to 25 times the dynamic power dissipated by a pipelined, but non-accelerated co-processor subsystem. Without pipelining we can invest *the same* amount of power into CO alone, but fail to meet the design target, because Fig. 6 only provides an upper bound. In other words: The factor of 5 in bus power mentioned above is certainly conservative. The true factor will indeed be greater than 5.

9. CONCLUSION

In terms of power efficiency, the pipelined operation of two subsystems is always superior to the non-pipelined variant, although we are using the same hardware. Insight into this matter is particularly important for signal processing in mobile terminals where battery power is naturally scarce. Savings in dynamic power dissipation mainly result from running the pipelined subsystems at different speeds, just as fast as needed to achieve the design goal.

The power efficiency potential may even be augmented by the fact that pipelining requires less bus connections per processor: The outlined arrows of Fig. 3 can all be left out, which reduces the hardware design effort and may as well bring down effective bus capacitances. Concrete values for the savings depend on the bus portion of the overall dynamic power dissipation.

We believe that pipelining, in general, shows great potential to become one of the major operating guidelines for Mod-SDR systems. So far, however, the advantages of pipelined operation have unexceptionally come at the price of additional memory demand and delay. These aspects have to be treated in more detail in future work on software partitioning for Mod-SDR systems.

10. REFERENCES

[1] B.W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol. 49, pp. 291-307, 1970.

[2] *IEEE Communications Magazine*, vol. 37, no. 2, Feb. 1999, special issue on Software Radio

[3] *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 4, April 1999, special issue on Software Radio

[4] W. Tuttlebee (Ed.), *Software Defined Radio: Enabling Technologies*, John Wiley & Sons Inc., 2002

[5] "Architecture and Elements of SDR Systems as Related to Standards", *SDR Forum Tech Report*, v2.1, Nov 1999, p.5-64

[6] ETSI TS 125 100/200 series, "Radio Transmission and Reception"/"Physical Layer", *ETSI*, v5.5.0 (2002-12)

[7] A.-R. Rhiemeier, F.K. Jondral, "Mathematical Modeling of the SDR Design Problem," *IEICE Trans. Commun.*, vol. E-86B, no. 12, Dec. 2003, special issue on SDR Technology

[8] S. Cho *et al.*, "Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems," *IEICE Trans. Commun.*, vol. E-85B, no. 12, Dec. 2002

[9] J. Mitola, "Software Radio Architecture," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 4, April 1999, pp. 514-538

[10] F.K. Jondral, "Parameter Controlled Software Defined Radio" in *Proc. SDR'02*, SDR Forum Technical Conference, vol. 2, sec. SY-2-02, Nov. 2002

[11] A.-R. Rhiemeier, "Benefits and Limits of Parameterized Channel Coding for Software Radio" in *Proc.2nd Karlsruhe Workshop on SDR*, ISSN 1616-6019, 2002, pp. 107-112

[12] A.-R. Rhiemeier, F.K. Jondral, "A Software Partitioning Algorithm for Modular SDR" in *Proc. WPMC'03*, Int'l Symp. Wireless Personal Multimedia Commun., Oct. 2003

[13] F.K. Jondral, "Parameterization – a Technique for SDR Impementation", in [4], pp. 232-256

[14] T. Hu, "Parallel Sequencing and Assembly Line Problems", *Operations Research*, vol. 9, pp. 841-848, 1961

[15] J.R. Sacha, M.J. Irwin, "Number Representations for Reducing Data Bus Power Dissipation", *Proc. Asilomar Conf. Sig. Sys. and Comp.*, vol. 1, Nov. 1998, pp. 213-21