Copyright © 2003, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

PERFORMANCE ANALYSIS OF THE PERIPHERAL – PROCESSOR INTERACTION IN EMBEDDED SYSTEMS

by

Christian Sauer, Matthias Gries, Chidamber Kulkarni
and Kurt Keutzer

Memorandum No. UCB/ERL M03/26

1 June 2003

PERFORMANCE ANALYSIS OF THE PERIPHERAL – PROCESSOR INTERACTION IN EMBEDDED SYSTEMS

by

Christian Sauer, Matthias Gries, Chidamber Kulkarni
and Kurt Keutzer

Memorandum No. UCB/ERL M03/26

1 June 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Performance Analysis of the Peripheral – Processor Interaction in Embedded Systems

Christian Sauer[1], Matthias Gries[2], Chidamber Kulkarni[2], Kurt Keutzer[2]

[1] Infineon Technologies, Corporate Research, Munich
[2] University of California, Berkeley
{sauer, gries, kulkarni, keutzer}@eecs.berkeley.edu

Abstract. *Designs of interconnection networks, memory hierarchy, and processors have received significant attention in the context of system-on-a-chip research. However, almost 30-40% of the on-chip real estate is spent on peripherals, which are heterogeneous in nature and have almost been neglected from a research perspective. Thus understanding issues that contribute to heterogeneity of peripherals is critical in making their design more homogeneous and regular. In this paper we present a study that models a complete embedded system including an operating system to understand the interplay between peripherals and the remaining system for different aspects. The results of benchmarking serial and infrared communication interfaces show that the protocol overhead handled at the peripheral-processor interface is at least 39% and between 16% and 60% of the effective system load are due to the peripheral. Our main observation based on this work is that a careful balance between local peripheral resources (buffer architecture, DMA policy, supported protocol stack functions) versus the granularity (number, frequency) and complexity (flow control) of system interaction is fundamental to optimizing peripheral-processor interaction (and making them homogeneous).*

Introduction

Rapid progress in embedded systems, such as Internet-attached devices, has given a new impetus to a wide and fast growing range of input-output (I/O) peripheral device standards such as USB, IrDA, IEEE 1394, and IEEE 802.11. This in turn has resulted in system architectures that combine customized blocks for peripherals with programmable processor cores.

Processors, memory hierarchies, interconnection networks, and peripherals are the main building blocks for these application specific programmable system-on-a-chip architectures. During the design process a lot of emphasis is put on the first three, but peripherals, including communication interfaces as their largest subset, are often neglected and their complexity is underestimated. This is mainly due to the general perception that peripherals constitute standard intellectual property (IP) blocks and that they are usually integrated at design time. Increasingly these peripherals are becoming bottlenecks to both the design process and the performance of the end

system, due to their heterogeneity and ad-hoc integration practice based on a localized view of the interfaces between processor and peripheral.

Our analysis of state-of-the-art chips [9] for multimedia and network domains clearly shows that 1) peripherals and communication interfaces account for a significant share of the total system, typically between 30% and 40% of the die area, 2) there is a large number of heterogeneous peripheral functions even on a single chip, and 3) a single peripheral can be more complex than a processor core. Finally, the software interface for such peripheral devices, which comprises device drivers and often the related protocol management, is becoming complex and more heterogeneous, resulting in a challenge for designing reliable software [11].

Thus any successful solution to the problem of heterogeneity, complexity, reliability, and ad-hoc integration methods necessitates a better understanding of existing mechanisms related to peripheral design and the corresponding system profile. Indeed, the primary goal of our research is to achieve a reduction in the diversity of peripherals thereby making the design process more regular and potentially simpler. The advantages of such a system are not only the unified design process, but also an increase in robustness of the overall system, due to a regular software interface for device drivers.

The main goal of this paper is to obtain a detailed understanding of the contribution of protocol-related overhead of peripherals to the utilization of the system and its components, and to reveal issues that contribute to the heterogeneity of such interfaces, showing the potential for optimization.

The related work for this paper is in three domains: those discussing the characterization and profiling of operating systems and protocol stacks, those addressing the synthesis of device drivers, and those addressing the interface and communication protocol synthesis problem.

In the domain of operating system and protocol stack characterization, the main focus is either on the impact of different processors on the operating system [7][8] or on the different timing profiles based on commercial benchmarks such as SPEC. Many of these works do not address the impact on device driver related issues. In the embedded domain there have been some works which investigate the impact on driver or peripheral related power consumption [1][12] but do not present a comprehensive picture on the impact of protocol stack and driver/peripheral aspects on bus, memory or even CPU. Apart from these works we find papers addressing device driver synthesis [5][14], which are concerned primarily with capturing existing driver/peripheral interfaces and encapsulating platform specifics so as to achieve portability. However, only little attention has been given to optimization of the overhead in performance introduced due to portability. Also the underlying interfaces remain unchanged. The synthesis of interfaces and communication protocols [6][10][13] has focused on formal models and automated techniques based on a localized view of the involved interfaces. In addition, there have been approaches that realize peripheral functions in software running on the core processor [4].

In summary, a lot of effort has been put into understanding meta-level operating system aspects, device driver and interface synthesis but existing work provides little insight into issues related to the optimization of lower-layers such as device drivers and peripheral – processor interaction. Thus, it is essential to study and investigate

these missing aspects to characterize the peripheral environment properly, so that the system integration and optimization can be eased and simplified.

In this paper, we model and evaluate a complete system comprising an ARM processor core, required controllers for interrupts and memory accesses, peripheral devices (UART, IrDA), and an embedded (uC-)Linux operating system.

The remaining paper is organized as follows: The next section briefly introduces trade-offs involved with the design of peripheral-processor interfaces. Section 3 presents the experiment system set-up and discusses different aspects of system adaptation. Section 4 presents and discusses the results obtained and identifies the bottlenecks in the total system due to device drivers and the related communication protocol. We conclude the paper with a summary of our main observations.

Interface Design Trade-offs

Multiple aspects characterize the peripheral - processor interaction. In particular, for an embedded system one needs to consider system load, communication overhead, interrupt overhead, and memory accesses due to the peripheral. The design space of a peripheral is quite complex, since we need to make many different trade-offs to address the above issues. Three of the main trade-offs are:

- Complexity of control in the peripheral (or related protocol stacks) as compared to the memory space and memory operations.
- Types of event handling mechanisms such as based on interrupts compared to those based on polling.
- Computational complexity of the processor as compared to the peripheral itself.

In addition, we also need to take into account software management aspects such as modularity of the system, robustness of the device driver interface, etc.

A better understanding of the peripheral – processor interaction is fundamental to the investigation of potential solutions to optimization of this interface. For this purpose we have assembled a framework that allows modeling a realistic embedded system in sufficient detail and also, lets users implement different optimizations. The goal of such an exercise is to enable experimentation with repartitioning of the functionality between peripherals and the processor, and the ability to adapt different interfaces for peripherals. In this paper we will focus on characterization and profiling of a realistic system to understand the above stated trade-offs.

System and Application

In this section we introduce the different aspects of the system, namely the simulation setup based on a processor simulator, the operating system, our application, and their adaptation for our needs. Our goal is to use a minimal but realistic embedded system

configuration that still exposes interfaces and their interaction with the environment sufficiently to characterize and identify bottlenecks.

We use an ARM based system [3] since the ARM processor family is well established and used in a wide variety of embedded systems. In addition, open source operating systems and public domain tool chains including simulators are available, which allows us to extend and customize the simulator and the OS with various peripheral modules.

For this paper, we implement interface functionality for two cases: a) Serial communication to consider the simplest possible way of connecting two systems and b) infrared (IrDA) communication due to its widespread use in handheld devices. Furthermore, these protocols provide two different implementations schemes: infrared is part of the operating OS/device driver whereas the serial communication requires a separate application.

File-Transfer application

File transfers, like the download of MP3 songs or e-mail synchronization, are examples for essential tasks of numerous embedded devices, especially in the domain of Internet-attached devices and handhelds. In order to expose the interaction between peripheral/communication interfaces and the remaining system, we have chosen a file transfer as benchmark since it is a small, but complete application and can be implemented on top of numerous protocol stacks such as USB, IrDA, and serial data links (UART).

For this study, a file is downloaded from a remote server into the system under observation. System and server are connected via their serial interfaces (see Figure 1). Kermit [2] is used as reliable protocol for the serial communication case whereas the IrDA stack guarantees reliable transmission in the infrared case. The workload of the communication interface not only depends on link properties (such as reliability) but also on the size of the file transferred. For this paper, we use synthetic files of sizes between 100 and 16000 Byte as workload and assume an unreliable link.

Architecture

We use an ARM processor as a CPU in our set-up. The system under test, as shown in Figure 1, is based on the AT91x40 series from ATMEL [3]. It contains an ARM7TDMi core with 4 MB RAM, 4 MB ROM, interrupt controller (AIC) and a set of peripherals, namely two timers (TC0/1) and two serial interfaces (UART0/1) with DMA capabilities (PDC).

We use an ARM simulator (ARMulator) with the GNU debugger (gdb) as our simulation backbone. We have modified the emulator to reflect the AT91 specific set of peripherals. The simulator accurately models instructions and their memory and I/O transactions and uses a flat memory model without data or instruction caches. The access to peripheral registers is memory mapped.

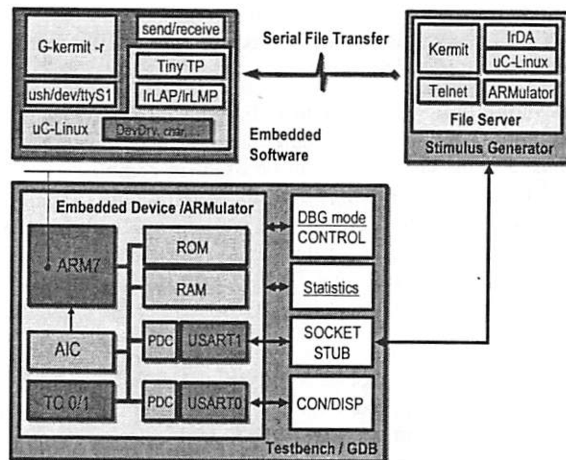

Fig. 1. Embedded System and Evaluation Environment.

Atmel's AT91x40 is designed to run with clock frequencies up to 40 MHz. Since the clock-per-instruction ratio for the ARM7 processor core is 1.9 and the transfer rate is up to 115 Kb/s for an UART, the available instruction budget lies somewhere between 361 (for 10 MHz) and 1460 Instructions (for 40 MHz) per transferred byte.

Embedded Software/Operating System

The operating system for our setup is uC-Linux[1], a derivative of the Linux 2.4 kernel that targets microcontrollers without a Memory Management Unit (MMU). The OS and its drivers can be customized to run on different processors and peripheral configurations. We have chosen a minimal configuration that contains only (beside the kernel and support for our devices) RAM/ROM disk support, a ROM based file system, loadable kernel modules, and a few core applications like msh, init, stty, ircpy, G-kermit and ush (see below).

System Adaptation

We had to implement the following crucial adaptations in order to make the complete set-up functional and useful in the context of peripherals as follows:

[1] www.uclinux.org

- **Additional peripheral** – The ARM system[2] needed to be extended with a second, more precise UART interface model, since the first simple one is only used for console and keyboard I/O of the system.
- **Distributed simulation** – In order to utilize the infrastructure of the simulation host the second UART interface in the emulator is mapped onto a TCP/IP port of the host. Thus, a telnet connection can be used (in combination with ush) for the file transfer. This operation mode is used for investigating the serial interface. For the infrared communication, another mode is used, which allows the connection of a second, identical system to the system under test. Both systems run in lock step and share the same simulation time. The TCP/IP based communication is solely part of the simulator backplane and fully transparent to the simulated interface and protocols.
- **IrDA support** – Linux device drivers for the infrared interface are still in experimental status and required some bug fixes to get corresponding sockets and utility packages to work in the ARMulator tool set. Based on the IrDA-Utils package, a bi-directional copy program was implemented (ircpy).
- **ARM applications** – Besides ircpy a shell (ush) that redirects I/O to the char device ttyS1 (our UART) was implemented. This enables a terminal connection to the device and thus allows applications to communicate via stdin/out with our system. The G-kermit program[3] uses this established link for reliable file transfers.
- **Debug mode control** – For the purpose of characterization a number of debug modules have been implemented in the emulator to closely watch program counter, memory accesses, the UART interfaces and the interrupt controller. Accessing reserved memory addresses via "gdb" commands can control these modules (enable/disable/reset/print).

System characterization and analysis

In this section we present a detailed system evaluation based on our set-up. The goal of our study is to understand the interaction between a peripheral and the surrounding system, especially the device driver. We therefore examine: 1) the communication overhead that needs to be handled by the peripheral for a particular protocol stack, 2) the profile of executed software and system load distribution to quantify the influence of the peripheral on computation and data movement, and 3) memory utilization and I/O access patterns for the memory mapped device and finally discuss our observations.

---

[2] www.uclinux.org/pub/uClinux/utilities/armulator/

[3] www.columbia.edu/kermit/gkermit.html

Protocol and communication overhead

For our application, the protocol overhead is introduced by G-Kermit[4] (in the serial communication case) or the IrDA protocol stack[5] (access IrLAP, management IrLMP, and transport TinyTP layers). Both protocols enable reliable packet-based transfers over a point-to-point connection.

The observable protocol overhead has five principal origins: a) **Accompanying data**: Besides the file content, descriptive information (like filename, type, and attributes) needs to be transferred. b) **Protocol initialization**: At the beginning of each session, parameters (e.g. packet length) need to be negotiated. c) **Packet frame**: Payload is encapsulated by header (start byte, packet length, and packet sequence number), and checksum. d) **Payload transcoding**: e.g. special control characters need to be escaped. e) **Flow control:** mechanism based on additional control packets (handshake, acknowledge). The latter three origins generate a permanent overhead, whereas the first two introduce only an initial, per session overhead.

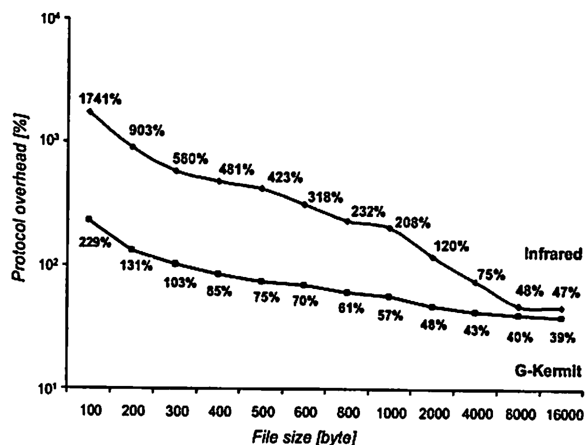

Fig. 2. Relative protocol overhead per payload size.

The amount of data that is passed through the UART peripheral is shown in Figure 2 for different payload sizes. Due to the mentioned initialization effects the overhead is high especially for small transfers, comprising up to 1740% (IrDA) of the payload size. Although with larger transfers the influence of initialization flattens and constant effects prevail, the overhead remains significant with 40% (Kermit) to 47% (IrDA) of the payload size. For unreliable communication links this overhead can be expected to

[4] www.columbia.edu/kermit/gkermit.html

[5] www.irda.org

be even higher because packet retransmissions and related link layer activities are necessary.

As expected, the relatively simple G-Kermit protocol defines a lower bound for the overhead. In the case of infrared communication, the situation becomes worse since additional protocol layers affect not only the amount of received data but also require more data sent back to the opposite layer. Downloading a 1000 byte file (as considered in the subsequent sections) requires 1570 bytes by serial or 3080 bytes by infrared to be transferred by the peripheral.

Software execution profile

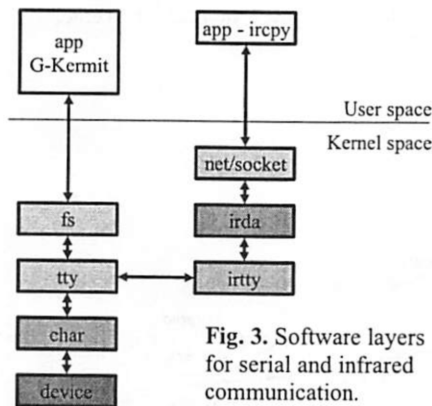

Fig. 3. Software layers for serial and infrared communication.

The function call stack for data transfers in our system has two origins/entry points: a) the peripheral device and b) the user application. In the first **hardware initiated** case, the UART receives data and generates an interrupt as soon as the data has been transferred into the main memory. The data is passed from there to the application (which independently calls the file system read) via different OS layers (char/tty driver – file system/fs – application; char/irtty – irda – net/socket – application) and a total of five different buffers. In the second **application initiated** case, a file system write is called that traps into the kernel and copies the data into another similar set of buffers to enable the transmission by the device driver's service routine.

In order to measure the system load, program counter (PC) traces are generated while executing the application with different workloads. These traces are used to compile histograms that summarize the system load in five to six classes with respect to call stack and OS layers[6]:

- **Application** – all user space applications, G-kermit or ircpy in our case.
- **Kernel** – all general system functionality: process scheduling, time handling, interrupt routines, signal interface; excluding system idle loop.
- **Idle** – non-functional. Although the idle loop technically belongs to the kernel a separate class helps identifying those non-functional instructions.
- **Char** – character device drivers. This class contains everything necessary to access our UARTs (n_tty, tty_io, atmel, random).

[6] Library functions called from one of the classes count always to the calling class, e.g. atmel/*receive_chars()* calls *memcopy*().

- **IrDA** – infrared protocol stack (access IrLAP, management IrLMP, and transport TinyTP layers) and ir_tty coupling to UART.
- **FS** – file system functions.

Figure 4 shows the relative distribution of executed instructions among our classes looking at the steady state for the transfer of a 1000 byte file. Initializations like load of application or UART setup/device discovery are not considered. The data is normalized to the total number of active instructions, i.e. without idle cycles, for speed independence.

We observe that in the **serial case** the interaction between peripheral and the remaining system across the char device driver accounts for an already significant share (16%) of the overall load. Furthermore, this driver is also the primary system aspect involved (followed by the interrupt handler (irq, softirq) 7%, timer and scheduler 5%), as Table 1 reveals. The application, however, accounts for 2/3rd of all executed instructions because of the protocol processing done at the application level. In the **infrared case**, the protocol layers are realized within the irda driver. Consequently, the irda driver is with 47% the largest share of the system load.

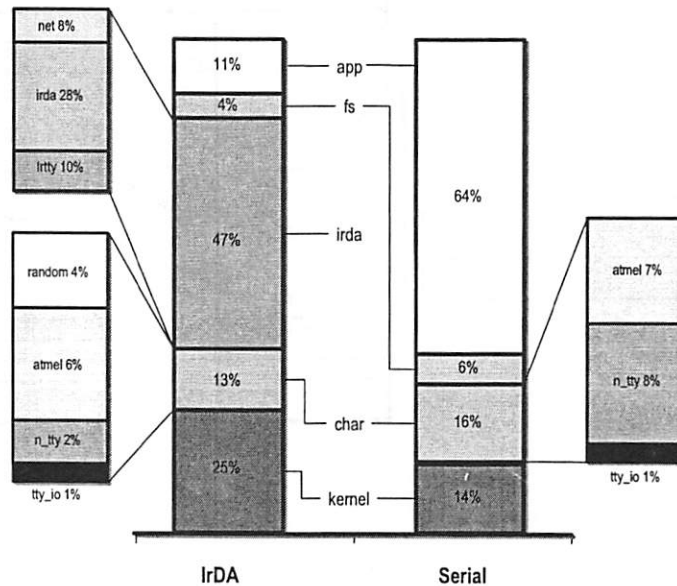

Fig. 4. Distribution of executed Instructions.

Tables 1 and 2 present the load per class and byte transferred in more detail, broken down by largest contributing functions. The number of 35 instructions per byte spent in the device driver (char) for the serial case is surprisingly high. This seems, at least in parts, due to multiple data movements. The six instructions per byte for the *receive_chars* function, for instance, only copy UART status & data between

buffers and schedule the next processing step. In the case of infrared we find a similar number of instructions (14.4 + 11.9) for comparable functionality spent in the char device driver and the irtty layer, respectively. The infrared protocol processing can be accounted with 34 instructions which are mainly (2/3rd) spent on frame processing.

Using the instruction budget (361 – 1460) for Atmel's AT91x40 in Section 0, we realize that: 1) 222 (118) non-idle instructions lead to a total system occupation between 15% (8%) and 61% (33%) for the serial (infrared) case, 2) the overall execution time is determined by the I/O speed only, and 3) the infrared transfer takes at least twice as long as the serial transfer at the same I/O speed because twice as much data and protocol overhead need to be transferred.

Table 1. Instruction count by function (serial case).

Task class		Largest functions	Instructions per byte		
char	n_tty	read_chan	12.4		
		n_tty_receive_buf	2.2		
		write_chan	1.4	17.2	
	atmel	receive_chars	6.2		
		rs_write	2.3		
		rs_interrupt	1.2		
		transmit_chars	0.8		
		start_rx	0.7	15.3	
	tty_io	tty_read	1.5	3.3	35.8
kernel	irq	do_IRQ	4.8		
		IRQ	3.6		
		check_irq_lock	1.5	12.8	
	time	...		7.4	
	sched	schedule	2.6	3.9	
	signal	sys_sigaction	1.7		
		do_sigaction	1.1	3.3	
	...				31.2
fs		sys_read	2.7		
		sys_write	1.5		13.1
G-kermit					142.7
	Total				222.8

Table 2. Instruction count by function (infrared).

Task class		Largest functions	Instructions per byte		
char	random	SHAtransform	3.7	4.3	
	n_tty	opost_block	1.5		
		write_chan	0.8	2.5	
	atmel	rs_write	2.2		
		receive_chars	1.3	6.5	
	tty_io	tty_write	0.7	1.0	14.4
irda	net	core/...	5.8		
		sched/...	2.8		
		socket/...	1.2	9.9	
	irda.o	state_inside_frame	9.9		
		stuff_byte	4.6		
		async_bump	3.6		
		irda_data_indication	2.6		
		state_begin_frame	2.2	34.2	
	irtty.o	irtty_receive_buf	6.6		
		irtty_change_speed_co...	4.6	11.9	56.0
kernel	sys_arm	sys_vfork	13.3	13.3	
	irq	do_IRQ	1.7	4.3	
		IRQ	0.8		
	mm	kfree	1.5	4.2	
		kmalloc	1.0		
	softirq	do_softirq	1.0	3.0	
	...			4.8	29.6
fs		sys_write	1.4		5.0
ircpy					12.9
	Total				117.9

Memory and I/O Accesses

The last part of this paper addresses memory accesses, especially those that involve our peripheral, the UART. The simulator not only accurately models instructions and their memory and I/O transactions but also those DMA accesses that are originated in the PDC/UART. Accesses to peripheral registers are memory mapped. Thus, the complete interaction with the UART will be reflected in the following.

For our benchmark (1000 Byte file transfer), in total, we recognize 196000 (serial) and 218000 (infrared) non-idle data accesses to the main memory, originated by the CPU (not counting any reads of instructions nor the peripheral DMA accesses). In

average, the memory is accessed two times for every three executed instructions (ratios: 0.56 serial, 0.6 infrared). Figure 5 shows the distribution of these accesses among our classes. In the serial case, we observe that just three sub classes (atmel, n_tty, and irq), account already for 55% of all OS memory accesses. The atmel serial driver is the most significant class. In the infrared case, the atmel driver still remains significant, but the irda driver class clearly dominates with 41%.

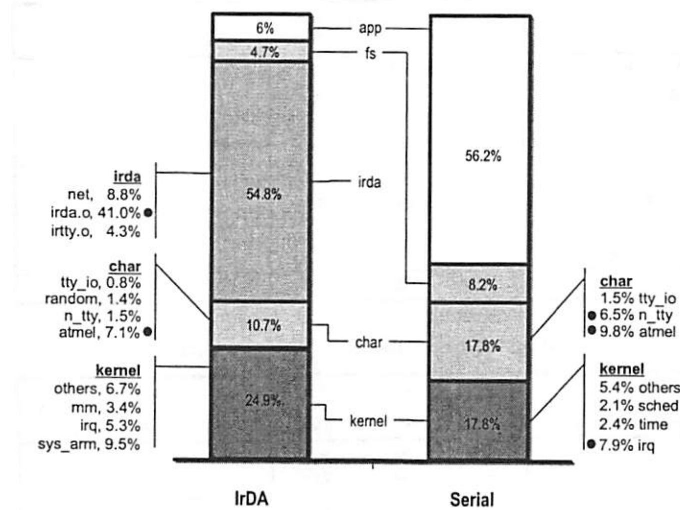

Fig. 5. Distribution of memory accesses among classes.

The CPU accesses the memory mostly in full words (88%, 81%), sometimes byte wise (12%, 17%), and only in a very few cases in half words. The memory mapped peripheral accesses are broken down by device in Table 3. The direct memory mapped CPU access to the UART [U1] is the reason for 7% (serial) and 4% (infrared) of all accesses initiated by *atmel*.

Table 3. I/O Accesses by Peripheral Device.

Peripheral device	Number of accesses – IrDA			Number of accesses – serial		
	Read	Write	Total	Read	Write	Total
Interrupt controller [AIC]	507	507	1014	294	364	658
Timer/Counter [TC]	102	0	102	18	0	18
Serial Interface [U1]	107	450	557	200	1135	1335
Total	716	957	1673	512	1499	2011

While the Timer device requires little maintenance, relatively much interaction is necessary for the serial interface. Looking closer into the UART, in Table 4, we realize that all accesses are directed to peripheral control registers. Data is transferred by the device' DMA controller directly to the memory and therefore, only pointer and

buffer length for such DMA must be set up. Interestingly, a sequence of 12 (average) control accesses must be executed per interaction because we count for the serial case 1335 accesses and a total of 109 events (91 receive interrupts and 18 transmit sequences, as shown later in Table 5). A peripheral interaction is either the reaction to DMA transfers plus subsequent interrupt (receive) or initiates (write data pointer) a transfer in order to transmit data (see Tables 4, 5).

Table 4. Control and Data Accesses for the UART peripheral.

Serial Interface [U1]		Number of accesses – IrDA			Number of accesses – Serial		
		Read	Write	Total	Read	Write	Total
DMA set up	Receivedata ptr	0	23	23	0	74	74
	Receive counter	46	23	69	91	148	239
	Transmit data ptr	0	15	15	0	18	18
	Transmit counter	0	75	75	0	127	127
Control	UART control	0	166	166	-	364	364
	INT enable/disable	-	46 + 58	104	-	91 + 165	256
	Channel status	61	-	61	109	-	109
	Receive Timeout	0	44	45	0	148	148
	Total	107	450	559	200	1135	1335

Table 5. UART1/memory DMA transfers.

Serial Interface [U1]	Bytes (IrDA)	Number (IrDA)	Av. Length [byte] (IrDA)	Bytes (Serial)	Number (Serial)	Av. Length [byte] (Serial)
Device – memory	2159	23	93.9	1385	91	15.2
Memory – device	926	15	61.7	141	18	7.8
Total	3085	38	81.2	1526	109	14

Although the DMA controllers could handle the full driver buffer size of 256 bytes, only 16 byte DMA transfers are at best executed in the serial case, as Table 5 reveals. Considering that the data is also moved into a special buffer (read, write) before the transfer is setup in a sequence of 12 cycles the usage of DMA is actually more expensive than writing directly to the device. The situation improves in the case of infrared. Here, the DMA is with an average transfer of 80 bytes much better utilized. Besides the amount of local memory, the length of a transfer depends only on the granularity of the used communication protocol. In our case in fact, the number of interactions equals the number of transmitted packets (23 received, 15 send).

Discussion

For the serial case, the char device driver, which handles the interaction between UART and processor/operating system, is not only the largest system aspect involved, but also accounts for a significant share of at least 16% of the system load and 18% of all memory accesses. Including the protocol processing as in case of IrDA, this numbers increase to 60% of the system load and 66% of all accesses. While this clearly confirms the successful separation of OS concerns, it also indicates the

necessity for the careful design of processor/OS - peripheral interfaces and the partitioning of tasks among peripheral and processor.

The assessment of Atmel's AT91x40 for our application shows, that the system would already be utilized between 8% and 61% by our application only. That means, 6 MHz of the processor speed are necessary to handle the 115kbps serial communication – a difference by a factor of 52. For the infrared communication, which needs twice as long, the factor is still 26. Although we consider only a minimal set of interfaces, this indicates arising problems for larger numbers of peripherals with potentially higher I/O bandwidth.

We find the number of 35 instructions spent per serially transferred byte within the char driver surprisingly high. It is caused partly by the movement of data across multiple buffers in different layers. Although the introduction of multiple driver layers is advantageous for abstraction and flexibility for programming both, user applications as well as drivers itself, we believe that optimizations across these layers are essential for an efficient implementation. This is supported by the somewhat better, but still high number of 22 instructions in case of the IrDA char driver implementation.

We find the measured communication overhead between 39% and 47% for larger transfers to be significant, especially, if the protocol stack as in case of Kermit is executed on top of the operating system, encompassing multiple layers and buffers. For unreliable connections packet retransmissions and related link layer activities may in addition require substantial system resources. Thus localizing the communication overhead to the peripheral by repartitioning the protocol stack is crucial. The IrDA protocol implementation within the device driver layers demonstrates already some relief for the higher OS/application layers compared to Kermit in the serial case.

We find the DMA capabilities within our system for the serial transfer underutilized, leading to additional bus congestion and communication delay; only a few bytes (15, 8) are transferred with each access, partly due to assumptions about the peripheral buffer size (NS PC16550 compatible mode) but also to the fine-grain structure of the communication. The IrDA protocol stack optimizes the first issue by deploying all local memory within the peripheral. In that way just a single interaction is required per infrared packet in our setup. The second issue, again, requires a different system partitioning that restricts fine grain feedback communication to the peripheral.

Approaches for Improving the Peripheral – Processor Interaction

The observations discussed in the previous section indicate clear potential for optimizing the processor – peripheral device interaction. In fact, we recognize three main approaches to this problem:

- **Interaction across software layers** – Most of the software layering is introduced for programming abstraction and flexibility/portability reasons and is not needed for a particular set of application and device. In our setup, for instance, 50% of the OS buffers and the related copy operations could be saved. Such optimization

however, requires a comprehensive/complete view onto application/OS/driver and hardware. Recent work addressing the customization of operating systems can be found in [15].

- **Processor support for fine grain I/O device interaction** – Standard processor architectures such as the ARM7 core provide only little support for frequent interrupts or excessive memory accesses. For communication-centric systems, potentially with multiple high bandwidth peripherals, deterministic scheduling and fast interrupt handling are however crucial tasks. Although achievable by over-provisioning, some hardware support such as multiple register contexts, or register mapped I/O could leverage more cost efficient systems. In that way, the observed discrepancy of a factor of 52 in required processor to peripheral speed could be reduced. Embedded network processors such as [16] already utilize such techniques.
- **Repartitioning of task distribution among processor and peripheral** – By repartitioning functionality among processor and peripheral, the designer can trade off local peripheral resources (buffer architecture, DMA policy, supported protocol stack functions) and the granularity (number, frequency) and complexity (flow control) of interaction. In this way, the system could be relieved from fine grain communication and also the protocol overhead – in our case between 39% and 229% for the serial transfer – would be kept local to the interface.

Conclusions

In this paper, we have modeled and evaluated a complete embedded system including operating system to profile and understand the interplay between peripherals and the remaining system. The main conclusions for our configuration are:

- The processor-peripheral interaction accounts for a significant share of system load, because of an unfavorable task partitioning between processor and peripheral.
- Communication-intensive and fine-grain device interaction is expensive for standard processors, because of little support for frequent context switches and I/O accesses.
- The interaction between the different layers of device drivers, operating system, and the application are highly suboptimal resulting in excessive memory usage/copying of data.

In summary, peripherals indeed contribute significantly to the overall system performance, and hence, a disciplined approach to the design and integration of peripherals is required, which is the focus of our current research.

References

[1] A. Acquaviva, L. Benini, B. Ricco, "Energy Characterization of Embedded Real-Time Operating Systems," *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Spain, 2001

[2] F. da Cruz, B. Catchings, "Kermit: A File Transfer Protocol for Universities", Part I+II, *Byte Magazine*, vol. 9, no. 6+7, USA, 1984

[3] Datasheet, Atmel AT91x40 series, www.atmel.com

[4] D. Lioupis, A. Papagiannis, D. Psihogiou, "A Systematic Approach to Software Peripherals for Embedded Systems," *Int. Symp. on Hardware/Software Codesign (CODES)*, Denmark, 2001

[5] M. O'Nils, A. Jantsch, "Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification," *Design Automation and Test in Europe (DATE)*, Germany, 1999

[6] R. B. Ortega, L. Lavagno, G. Borriello, "Models and Methods for HW/SW Intellectual Property Interfacing," *ASI Proceedings on System Synthesis*, Italy, 1998

[7] J. Redstone, S. Eggers, H. Levy, "An Analysis of Operating Systems Behavior on a Simultaneous Multithreaded Architecture," *9th ASPLOS Conference*, 2000

[8] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, A. Gupta, "The Impact of Architectural Trends on Operating System Performance," *15th ACM Symposium on Operating Systems Principles (SOSP)*, Colorado, USA, 1995

[9] C. Sauer, "Modeling Peripherals," Fully Programmable Systems Theme, *GSRC Workshop*, Stanford, CA, 2002.

[10] J. Smith and G. De Micheli, "Automated Composition of Hardware Components," *35th Design Automation Conf. (DAC)*, 1998

[11] R. Short, B. Stuart, "Windows XP crash data," Driver Development Keynote, *WinHEC*, Anaheim, USA, 2001

[12] F. Vahid, J. Henkel, "Instruction based System-level Power Evaluation of System-on-a-Chip Peripheral Cores," *13th Int. Symp. on System Synthesis (ISSS)*, Spain, 2000

[13] S. Vercauteren, B. Lin, "Hardware/Software Communication and System Integration for Embedded Architectures," *Kluwer Journal on Design Automation for Embedded Systems*, vol. 2, no. 3/4, 1997

[14] S. Wang, S. Malik, R. Bergamaschi, "Modeling and Integration of Peripheral Devices in Embedded Systems," *Design Automation and Test in Europe (DATE)*, Germany, 2003

[15] J. Navarro, "OS and Software Stack Customization," Embedded Software Theme, *GSRC Workshop*, Anaheim, CA, 2003

[16] T. Halfhill, "Ubicom's new NPU stays small," *Microprocessor Report*, April 21th 2003