Profiling Bluetooth and Linux on the Xilinx Virtex II Pro

Filipa Duarte and Stephan Wong
Computer Engineering
Delft University of Technology
{F.Duarte, J.S.S.M.Wong}@ewi.tudelft.nl

Abstract

In this paper, we present profiling results of the Bluetooth standard implemented on the Xilinx Virtex II Pro device. The investigation is performed in two stages. First, we solely focus on the Bluetooth standard and its internal functions. Second, we focus on the Bluetooth standard in conjunction with an operating system. In both stages, we determine the most time-consuming functions by pinpointing the most computationally intensive or most data-intensive functions. The results of the first stage show that there are eight most time-consuming functions within Bluetooth. The results of the second stage show that (excluding interrupt related functions) the most time-consuming function is memcpy. *This function is called inside of the Bluetooth standard in order to move received packets to another memory location (allowing new packets to be copied to the same memory location) and to reassemble a frame with all the received packets.* memcpy *is also called within other standard Linux networking protocols which should similarly benefit from an hardware implementation of this function.*

1 Introduction

Nowadays, we are witnessing a continued and persistent trend to integrate a multitude of functionalities into a single device. This is, in particular, true for mobile devices in which agenda functionalities, music, games, and wireless connectivity are integrated. Depending on the application at hand or the available wireless network surrounding the device, multiple communication standards can be used, e.g., wireless LAN (Wi-Fi), GPRS, UMTS, Bluetooth, etc.

The traditional approach to achieve this integration is twofold. First, more powerful general-purpose processors (GPPs) can be used to meet the performance requirements of the newly added functionalities. Second, different application-specific processors (ASPs) can be used to augment an existing GPP and perform the most computationally or data intensive parts. It must be noted that both ways of integration are not mutually exclusive and can indeed complement each other to achieve better integration results. Furthermore, it is generally accepted that the extended GPP is more flexible but it lacks in performance. On the other hand, ASPs provide adequate performance but usually lack in flexibility. Finally, the GPP is usually used to perform more control-related tasks, while the ASPs perform more computationally intensive tasks. Figure 1 depicts this traditional approach.

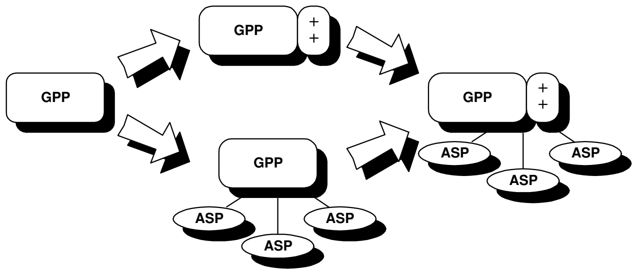

Figure 1. Traditional approach

This straightforward manner of integration is bound to lead to bulkier, heavier, and more power hungry devices simply due to the inclusion of the multitude of chips or on-chip functional blocks. An alternative solution is to implement the most computationally and data intensive tasks of an application (as well as other frequently used hardware designs) on a reconfigurable processor (RP), and to leave other tasks (usually control related) to be performed by the GPP. An example approach of augmenting an RP to a GPP is described in [11]. Figure 2 depicts a simplified view of this approach.

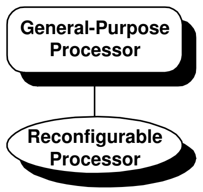

Figure 2. Reconfigurable approach

A requirement for utilizing the RP is that it must be large enough to perform the tasks that need to be executed simultaneously or that do not have to be performed concurrently (in turn, requiring a smaller reconfigurable processor). The added benefit of utilizing an RP is that it allows for increased performance (for those supported tasks) utilizing special hardware designs and increased flexibility (changing functionality over time) using a single RP. Other advantages include the reduction of the overall cost of the final product and the implementation of future applications without additional hardware. Moreover, the time to market can be much reduced.

Like the traditional utilization of an ASP, combining a GPP with an RP requires the determination of the most compute/data-intensive operations through profiling of the application or networking standard in question. As a starting point for our research, we investigate the Bluetooth standard. Our results show eight functions as the most time-consuming ones, when purely looking to the processing within the Bluetooth standard. The investigated Bluetooth implementation is also tightly intertwined with the Linux operating system (OS). Consequently, our profiling results include OS functions being called in the Bluetooth processing. When including the OS functions (but excluding the interrupt related functions) the most time-consuming function is *memcpy*.

The paper is organized as follows. In Section 2, we present related work and in Section 3 we introduce the Bluetooth stack and one of its implementation. In Section 4, we describe the systems and the set-up used to obtain the profiling information. In Section 5, we present and discuss the profiling results both related to the Bluetooth standard alone and including the operating system functions. We identify the candidate function(s) to be implemented on the RP and finally, in Section 6 we draw some conclusions.

2 Related Work

To our best knowledge, no profiling information on the Bluetooth protocol has been published. Therefore, we present in this section related work on profiling other related network protocol stacks, more specifically on TCP/IP and UDP/IP. Finally, we describe some related work on the implementation of an operating system (most times including network functionality) onto reconfigurable hardware platforms.

In [4] and [7], it is concluded that the main performance bottlenecks in TCP/IP and UDP/IP protocol processing are the checksum calculations and data movements. The same conclusion is reached in [8] and [13] with the latter including the impact of the operating system on the communication system. The authors present in their survey papers several solutions to solve those bottlenecks. Those solutions include (1) offloading checksum calculations (to solve the checksum bottleneck), (2) using direct memory access (DMA) instead of programmed Input/Output (PIO), and (3) avoiding cross-domain data transfers through 'zero-copying' schemes (to solve the data movement bottleneck). The Bluetooth protocol, as a communication protocol that rely on TCP or UDP over IP, can likewise benefit from the same techniques. In addition, the underlying layers (specific layers from the Bluetooth protocol) perform data movement, as discussed later.

In our investigation, we utilize a reconfigurable hardware platform to profile the Bluetooth standard. Consequently, we consider the porting of operating systems to reconfigurable hardware platforms as related work. In [14] and [15], the authors present their fundamental set of services that should be providing by an OS for reconfigurable computing: resource allocation and partitioning, application placement, routing, loading and scheduling, and a need for an appropriate communication abstraction between the OS and the application. Their proposed OS is Java-oriented. In [10], the authors focus on similar issues by presenting extensions to an existing real-time OS and concentrating on the reconfigurable computing issues since the support for regular software tasks is already present on the existing OS. In [12], a run-time partial reconfiguration environment is presented. It primarily functions as an OS executing concurrent hardware blocks instead of conventional sequential processes. They propose robust and flexible platforms to design and execute hardware at an object-oriented software abstraction level. Finally, in [18] the SHUM-uCOS framework is presented, which is an extended version of an existent real-time OS. It makes the differences between hardware and software transparent for programmers, but, as an OS, it is aware of those differences. This OS is based on a multi-task model and is applicable to fully configurable FPGAs (Field-Programmable Gate Array).

3 The Bluetooth Standard

The Bluetooth standard was primarily designed to substitute wires. Over time, Bluetooth developers and users have identified other applications that can benefit from Bluetooth. These include, e.g., LAN access points, file transfers, etc. Bluetooth allows a maximum of 7 devices in a range of 100 meters (for the specifications v1.2, see [2]; for a general overview of the protocol, see [1]) to communicate with each other and/or with a master device, with a maximum throughput of 723.2 Kbps.

The wide range of possible Bluetooth applications implies that there are many Bluetooth software layers. The lower layers (Radio Baseband, Link Controller, and Link Manager) are very similar to any other over-air transmissions. They can handle error detection and re-transmission, and manage the links between devices. They can also provide voice connections and a single data pipe between two or more Bluetooth devices. To ease integration of Bluetooth into existing applications, the Bluetooth specification provides middle layers that attempt to hide some of the complexities of wireless communications.

The fundamental layers of Bluetooth wireless technology (see Figure 3) are: Radio Baseband, Link Controller and Manager, Logical Link Control and Adaptation Protocol (L2CAP), and Service Discovery Protocol (SDP). On top of these layers (excluding the SDP), different applications require different selections from these higher layers

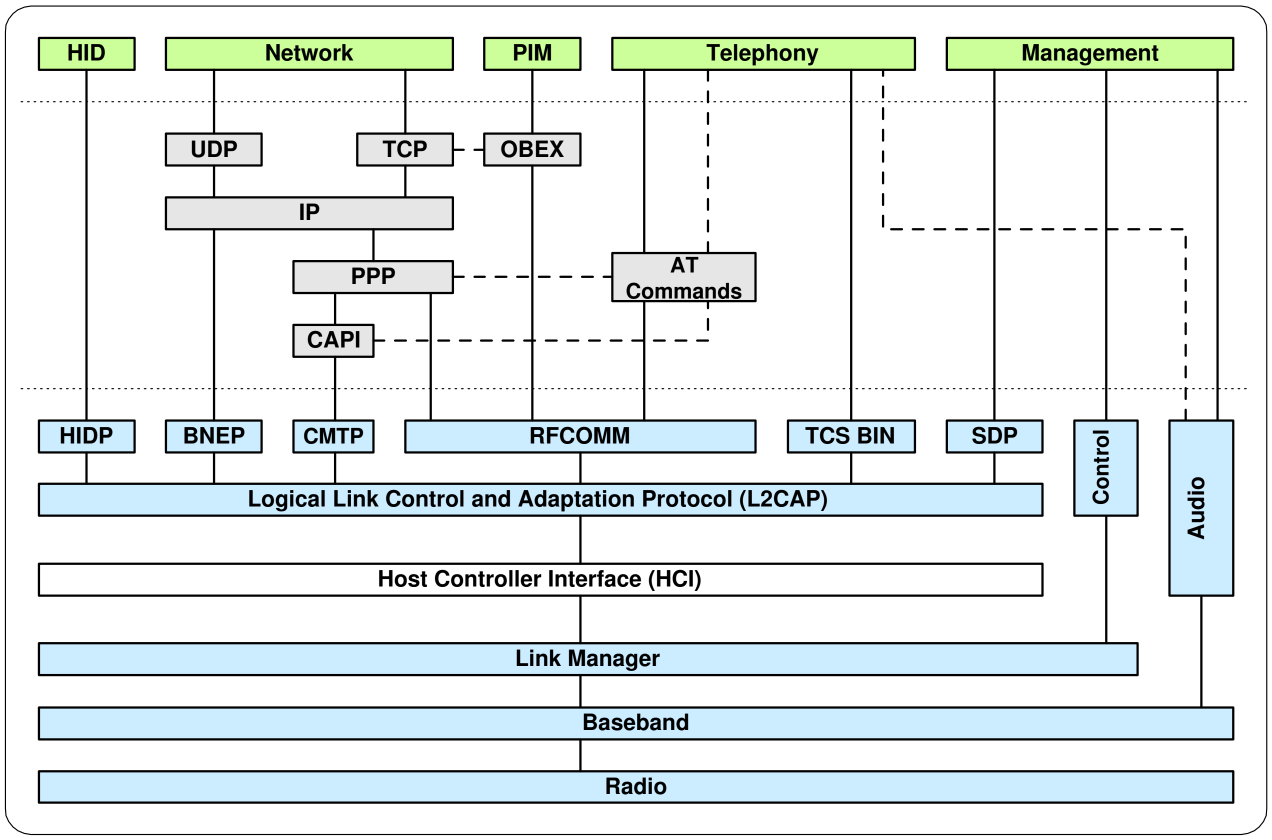

Figure 3. Bluetooth Stack Layers

(see Figure 4). Each profile (or application) calls the higher layers it requires.

The lower stack layers reside on the Bluetooth dongle, while the upper stack layers resides on a host (this may be a PC or a micro-controller if the product is stand-alone or mobile). The Bluetooth dongle and the host communicate via the Host Controller Interface, which is located between the lower layers and upper layers of the protocol stack forming a bridge between them. HCI is not a software layer, but a transport and communications protocol that aids interoperability between different manufacturers solutions. A short summary of the layers is presented in the following.

The L2CAP layer multiplexes upper layer data onto the single Asynchronous Connection-Less (ACL) connection between two or more devices and, in the case of a master device, directs data to the appropriate slave. It also segments and reassembles the data into chunks that fit into the maximum HCI payload. Locally, each L2CAP logical channel has a unique Channel Identifier (CID) and all channels, apart from broadcasts (transmissions from a master to more than one slave simultaneously), are considered reliable. The stack layers that are located above L2CAP can be identified by a Protocol Service Multiplexor (PSM) value. Remote devices request a connection to a particular PSM, and L2CAP allocates a CID. There may be several open channels carrying the same PSM data.

RFCOMM (a name stemming from an Radio Frequency-oriented emulation of the serial COM ports on a PC) emulates a full 9-pin RS232 serial communication over an L2CAP channel. A master device must have separate RFCOMM sessions running for each slave requiring a serial port connection.

The SDP layer differs from all other layers above L2CAP in that it is Bluetooth-centered. It is not designed to interface to an existing higher layer protocol, but instead addresses a specific requirement of Bluetooth operation: finding out what services are available on a connected device. The SDP layer acts like a service database. The local application is responsible for registering available services on the database and keeping records up to date. Remote devices may then query the database to find out what services are available and how to connect to them.

The Human Interface Device (HID) protocol defines a set of services for using HID devices (such as keyboard and mouse) over a Bluetooth connection. Before activating HID, it must send its device information to the driver on the host. After, a Bluetooth connection is established between the HID and the host, and the HID can then communicate with the driver over an L2CAP connection.

Bluetooth devices can also communicate with each other by using standard networking protocols to transport control and data packets over a network. Devices may use pro-

Profile	Lower Layers	L2CAP	SDP	RFCOMM	HID	BNEP	CMTP	TCS
Service Discovery Application	X	X	X					
Sinchronization	X	X	X	X				
FAX	X	X	X	X				
File Transfer	X	X	X	X				
Generic Object Exchange	X	X	X	X				
Headset	X	X	X	X				
Serial Port	X	X	X	X				
Intercom	X	X	X					X
Common ISDN Access	X	X	X				X	
Personal Area Network	X	X	X			X		
Human Interface Device	X	X	X		X			

Figure 4. Layers involved in each profile

tocols such as TCP, IPv4 or IPv6. These protocols have dissimilar network packet formats. To provide seamless transmission of network packets over the L2CAP layer in the protocol stack, an intermediate protocol is required that encapsulates dissimilar network packet formats as a standard common format. The Bluetooth Network Encapsulation Protocol (BNEP) provides this encapsulation by replacing the networking header, such as an Ethernet header, with a BNEP header. The L2CAP layer then encapsulates the BNEP header and the network payload and sends it over the transport media.

The CAPI Message Transport Protocol (CMTP) is used to transfer Common ISDN Application Interface (CAPI) messages. The data is transferred via L2CAP. To achieve the use of maximum Bluetooth bandwidth it is reasonable to concatenate several messages in one L2CAP data block. For data delay reason, it is sensible to transport the data as quickly as possible. Therefore, data should be sent immediately after reception from the ISDN line. In case of a receive error, the partly transferred blocks can be discarded by setting an error value in the last part of the message. Besides concatenation of CAPI messages as stated above, CAPI messages may be split over several CMTP messages. CMTP messages shall not be split over multiple L2CAP packets. Parts of different CAPI Messages may be interleaved by using different CMTP Block IDs.

The Telephony Control Protocol Specification Binary (TCS-Bin) includes a range of signaling commands from group management to incoming call notification, as well as audio connection establishment and termination.

As the Bluetooth standard performs different tasks (applications or profiles), it is a multitasking system, therefore it requires a scheduler or an Operating System (OS). There is some obvious synergy if we look to Bluetooth technology and Linux OS. Bluetooth is an open standard while Linux is open source. This allows to combine low cost devices with free software. Only recently, the Linux kernel included the Bluetooth stack among its stock drivers. In 2001, a Bluetooth project for Linux was released as open source and rapidly accepted into the 2.4.6 kernel. This project is called BlueZ [3], and it includes stable HCI, L2CAP, and RFCOMM drivers, as well as user-space SDP applications.

4 Measurement Environment

We installed in two systems the Linux OS and the BlueZ stack in order to retrieve the profiling information of the Bluetooth standard. The systems used to perform the experiments are the following:

- A desktop Intel Pentium 4 (at 2.80 GHz) running Linux 2.4.22.
- The Xilinx ML310 Embedded Development Platform [17] with a Virtex II Pro FPGA containing two Power PCs 405 (at 300 MHz). The Linux MontaVista 2.4.24 [9] is running only on one of the Power PCs.

One of the most interesting applications (profiles) that can be executed over Bluetooth is 'file transfer'. This profile will imply the use of the lower layers, L2CAP and BNEP as well as SDP. In our case, as we are using the Bluetooth USB adapter from Conceptronic [5], the HCI layer will also be needed to transfer the required information between the lower and upper layers of the Bluetooth stack. As expected, 'file transfer' will also utilize part of the TCP/IP stack of Linux. In order to do a 'file transfer', we created a file of 50 Megabytes (MB), that we transfer between the Bluetooth devices.

The profiler used to retrieve the profiling information is part of the Linux kernel, and the application that is able to interpret that information is also a kernel build-in application, *readprofile*. To identify the Bluetooth functions, the Bluetooth stack has to be statically linked into the ML310 platform Linux kernel. As we are dealing with an implementation of Bluetooth on the Linux OS we need to take into account that the OS is not a deterministic application. Depending on the state of the system, the OS will react differently. Therefore, in order to retrieve meaningful data, we performed 20 identical and independent trials of the same experiment. The trials were performed without rebooting the system and the profiler was restarted for each trial.

5 Profiling Results

In Table 1, the first 40 rows of the profiling information of one of these trials, sorted by number of ticks (N.Ticks), are presented. The N.Ticks is a value returned by the profiler representing the number of times a particular function was running on the processor when the system was interrupted by the profiler to retrieve the profiling information.

N.Ticks:	Function Name:
69274	__sti_end
2247	__sti
1136	__save_flags_ptr_end
153	memcpy
150	hc_interrupt
138	speedo_interrupt
127	trident_interrupt
126	XSysAce_RegRead32
81	sohci_submit_urb
68	__copy_tofrom_user
67	XSysAce_RegWrite32
64	dl_done_list
62	hci_usb_rx_complete
61	l2cap_recv_acldata
61	dl_transfer_length
59	tcp_recvmsg
52	hci_usb_rx_submit
48	sys_select
48	kmalloc
47	hci_send_to_sock
45	__kfree_skb
44	hci_rx_task
43	skb_under_panic
41	fget
39	normal_poll
38	do_select
37	__kmem_cache_alloc
36	DoSyscall
35	tty_poll
35	power_save
35	__save_flags_ptr
33	__free_pages_ok
32	tcp_rcv_established
31	hci_recv_frame
31	bnep_rx_frame
30	tcp_poll
30	hci_acldata_packet
29	tasklet_action
28	tcp_v4_rcv

Table 1. The first 40 rows of the profiler sorted by number of ticks

Normalized Load:	Function Name:
2164.8125	__sti_end
28.0875	__sti
11.3600	__save_flags_ptr_end
1.5000	XSysAce_RegRead32
1.2885	XSysAce_RegWrite32
0.9808	memcpy
0.6029	fget
0.6000	kmalloc
0.5040	trident_interrupt
0.4375	__save_flags_ptr
0.3913	DoSyscall
0.3074	hc_interrupt
0.3017	power_save
0.2315	speedo_interrupt
0.1906	dl_transfer_length
0.1862	tty_poll
0.1616	dl_done_list
0.1449	hci_usb_rx_complete
0.1318	tasklet_action
0.1310	hci_rx_task
0.1230	hci_recv_frame
0.1206	__copy_tofrom_user
0.1160	__kfree_skb
0.1066	hci_usb_rx_submit
0.1048	normal_poll
0.0934	__kmem_cache_alloc
0.0882	hci_acldata_packet
0.0822	hci_send_to_sock
0.0786	l2cap_recv_acldata
0.0714	do_select
0.0714	tcp_poll
0.0694	skb_under_panic
0.0671	sohci_submit_urb
0.0441	sys_select
0.0326	__free_pages_ok
0.0280	bnep_rx_frame
0.0273	tcp_recvmsg
0.0180	tcp_v4_rcv
0.0149	tcp_rcv_established

Table 2. The first 40 rows of the profiler sorted by Normalized Load

Table 2 presents the first 40 rows sorted by Normalized Load, which is another value returned by the profiler. It is calculated by dividing the number of ticks recorded by the profiler for a function by the function size (i.e., length of the address space in memory)[16]. Therefore, it is safe to assume that functions rating high in both lists (Table 1 and 2) occupy the processor and are, therefore, the most time-consuming.

From Tables 1 and 2, we can identify eight Bluetooth functions: *hci_usb_rx_complete*, *l2cap_recv_acldata*, *hci_usb_rx_submit*, *hci_send_to_sock*, *hci_rx_task*, *hci_acldata_packet*, *hci_recv_frame* and *bnep_rx_frame*, and a mixture of OS and TCP/IP functions. Moreover, we can identify that, in the majority of time, the processor is controlling interrupts (e.g., *__sti_end*, *__sti*, which occupied 95% of the time) or their execution (e.g., *hc_interrupt*, *speedo_interrupt*, *trident_interrupt*) or is copying data within memory (e.g., *memcpy*). The interrupt-related functions are the most time-consuming ones, however, they are less good candidates to be implemented on a reconfigurable processor. The reason for this is because each time an interrupt is issued, the way the OS deals with it is depended of the state of the system. Therefore, the executed code may be slightly different each time the same interrupt is issued.

A comparison of the number of ticks of the Bluetooth functions and the *memcpy* function is depicted in Figure 5. It is clear that *memcpy* is more time-consuming than any other Bluetooth function.

The *memcpy* function is responsible for copying data of size *count* from address *src* to *dest* (the C code is presented below). It is worth noting that there has been extensive on-going research on how to optimize this function in software. The most utilized solution is to write this function manually

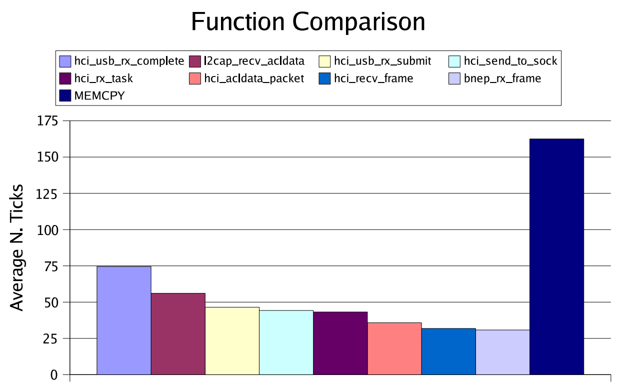

Figure 5. Comparison of the number of ticks of the Bluetooth functions and *memcpy*

in assembly and link it to the OS instead of compiling the C code. This will result in more efficient code.

```
/**
 * Copy one area of memory to another
 * @dest: Where to copy to
 * @src: Where to copy from
 * @count: The size of the area.
**/
void *memcpy(void *dest,const void *src,
             size_t count)
{
  char *tmp=(char *)dest, *s=(char *)src;

  while (count--)
     *tmp++ = *s++;

  return dest;
}
```

In order to understand why the *memcpy* function is one of the most time-consuming, we annotated the Bluetooth functions within of the Linux kernel in order to determine which functions were calling it. We identified four actions performed by the kernel when handling a ‘file transfer’: ‘frame acknowledging’, ‘interrupt handling’, ‘receiving packet’ and ‘reassembling frame’. A graphical view of these actions is depicted in Figure 6. The functions that appear on the top of the profiling information are part of the ‘receiving packet’ action, which is repeated until it receives enough data to reassemble a frame. The ‘interrupt handling’ action can occur any time during the execution of either other actions.

Besides annotating the functions, we also annotated the *dst*, *src* and *size* of the Bluetooth functions that call the *memcpy* function (i.e., *hci_usb_interrupt*, *l2cap_recv_acldata* and *bnep_rx_frame*). We identified a repetition on the *dst* address when receiving a packet and reassembling a frame. The *size* of the copies is also regular (being 339 bytes for

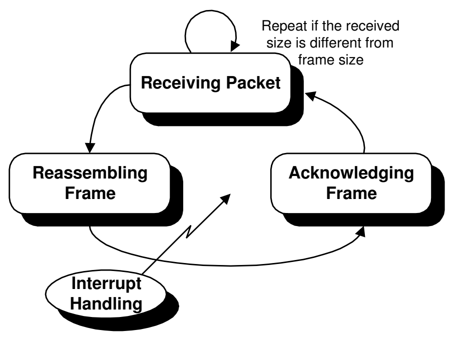

Figure 6. Bluetooth Actions

four packets plus 151 bytes for the last packet on the ‘receiving packet’ action and 1507 bytes when ‘reassembling frame’).

The analysis presented above only considers the influence of *memcpy* on the Bluetooth standard. However, as mentioned in Section 2, data movement is a bottleneck on the TCP or UDP over IP protocol stack. The profiling information obtained for *memcpy* (presented in Table 1 and 2) include the Bluetooth functions presented above plus the OS and TCP/IP functions. As such, *memcpy* is the most likely candidate function to be implemented on the reconfigurable processor.

6 Conclusions

We have observed a trend in integrating an increasingly large number of functionalities in a single device. This trend implied the use of general-purpose processors often augmented with application-specific processors in order to achieve the required performance. Currently, the trend is progressing towards the inclusion of a reconfigurable processor, substituting the multitude of application-specific processors. In order to use a reconfigurable processor (as well as a combination of a general-purpose processor augmented with an application-specific processor) profiling information about the applications is needed. In this paper, we focused on the Bluetooth standard. We presented profiling information related to a ‘file transfer’ application between two Bluetooth devices. Our results showed eight functions as the most time-consuming ones, when purely looking to the processing within the Bluetooth standard. Taking into account the OS functions (but excluding the interrupt related functions) the most time-consuming function is *memcpy*. As such, it is the function proposed to be implemented on a reconfigurable processor.

References

[1] P. Bhagwat. Bluetooth: Technology for short-range wireless apps. *IEEE Internet Computing*, pages 96–103, May–June 2001.

[2] Bluetooth specification v1.2.
`http://www.bluetooth.org/spec/`.

[3] BlueZ.
`http://www.bluez.com`.

[4] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP Processing Overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[5] Bluetooth USB Adapter.
`http://www.conceptronic.net`.

[6] D. Kammer, G. McNutt, B. Senese, and J. Bray. *Bluetooth Application Developer's Guide: The Short Range Interconnect Solution*. Syngress Publishing, Inc., 2002.

[7] J. Kay and J. Pasquale. Profiling and Reducing Processing Overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, pages 817–828, December 1996.

[8] S. Makeni and R. Iver. Performance Characterization of TCP/IP Packet Processing in Commercial Server Workloads. In *Proceedings of the 2003 IEEE International Workshop on Workload Characterization*, pages 33–41, 2003.

[9] MontaVista Linux.
`http://www.mvista.com`.

[10] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, 2003.

[11] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Painainte. The MOLEN Polymorphic Processor. *IEEE Transactions on Computers*, pages 1363–1375, November 2004.

[12] V.Groza and R. Abielmona. What next? A Hardware Operating System? In *Instrumentation and Measurement Technology Conference*, pages 1496–1501, 2004.

[13] P. Wang and Z. Liu. Operating System Support for High-Performance Networking, A Survey. *Journal of China Universities of Posts and Telecommunications*, pages 32–42, September 2004.

[14] G. Wigley and D. Kearney. The Development of an Operating System for Reconfigurable Computing. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 249–250, 2001.

[15] G. Wigley and D. Kearney. Research Issues in Operating Systems for Reconfigurable Computing. In *Proceedings of International Conference on Engineering Reconfigurable Systems and Architecture*, 2002.

[16] M. Wong. Stressing Linux with Real-World Workloads. In *Proceedings of the Linux Symposium*, pages 495–504, 2003.

[17] Xilinx ML310 Development Platform.
`http://www.xilinx.com/products/boards/ml310/current`.

[18] B. Zhou, W. Qiu, and C. Peng. An Operating System Framework for Reconfigurable Systems. In *Proceedings of the Fifth International Conference on Computer and Information Technology*, 2005.