Bluetooth protocol profiling on the Xilinx Virtex II Pro

Filipa Duarte and Stephan Wong

Computer Engineering Laboratory, Delft University of Technology

{F.Duarte, J.S.S.M.Wong}@ewi.tudelft.nl

Abstract—**Nowadays, there is an increasingly stronger trend to integrate a multitude of functionalities into a single device. Traditionally, this has been achieved by utilizing more powerful general-purpose processors to handle the additional workload. Since then, application-specific processors (acting as co-processors or hardware accelerators) were introduced to offload part of these workloads and to more efficiently perform them (e.g., in terms of speed power consumption). Currently, the trend is progressing towards the inclusion of reconfigurable processors to perform those workloads that do not need to be performed at the same time. In this manner, foremost costly area can be saved and less application-specific processors need to be designed and/or included. In the latter two approaches, application profiling is needed to determine compute-intensive or data-intensive operations to be implemented in specialized hardware. In this paper, we present the profiling results of an implementation of the Bluetooth standard. To our knowledge it is the first time profiling results of the Bluetooth standard are presented. First, we solely focus on the Bluetooth standard and determine the most compute/data-intensive operations/functions. Second, we investigate and consider external functions that are called by the Bluetooth code due to its tight integration within the operating system. Our results show that six Bluetooth functions are the most intensive ones. When including the operating system functions (but excluding the interrupt related functions), the most intensive function is *memcpy*.**

Keywords— **wireless protocols, Bluetooth protocol, profiling, reconfigurable hardware**

I. Introduction

Nowadays, we are witnessing a continued and persistent trend to integrate a multitude of functionalities in 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 network surrounding the device, multiple 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 (GPP) can be used to meet the performance requirements of the newly added functionalities. Second, different application-specific processors (ASP) 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.

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. An alternative solution is to implement the most computation-
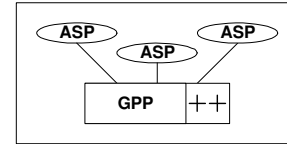

Fig. 1. Traditional approach

ally and data intensive tasks of an application (as well as other frequently used hardware designs) on a reconfigurable processor (RP), and leave other tasks (usually control related) to be performed by the GPP. Figure 2 depicts a simplified view of this approach.
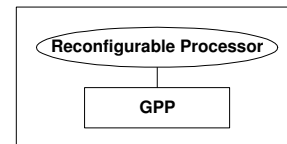

Fig. 2. Reconfigurable approach

A requirement for utilizing a 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 Bluetooth standard, because as of date, no profiling information on Bluetooth has been presented. Our results show six functions as the most intensive 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 intensive function is *memcpy*.

The paper is organized as follows: Section II presents

the Bluetooth stack and one of its implementation. In Section III, we describe the systems and the set-up used to retrieve the profiling information. In Section IV, we present and discuss the profiling results related the Bluetooth standard and in Section V we present and discuss the results when including the operating system functions, and we identify the function(s) candidate to be implemented on the RP. Finally, in Section VI we draw some conclusions.

II. The Bluetooth Standard

The Bluetooth standard was primarily designed to substitute wires. Besides that, Bluetooth developers and users have identified, over time, other applications that Bluetooth can be suited for. These may include, e.g., LAN Access Points, File Transfers, etc. Bluetooth allows a maximum of 7 devices in a range of 100m (for the specifications v1.2 [1]) to communicate with each other and 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, different applications require different selections from these higher layers (see Figure 4). Each profile (or application) calls up the higher layers it requires.

The lower stack layers reside on the Bluetooth dongle, while the upper stack resides on a host (this may be a PC or a micro-controller if the product is mobile or standalone). 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.

The L2CAP layer multiplexes upper layer data onto the single Asynchronous ConnectionLess (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). L2CAP only deals with data traffic, not voice, 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,
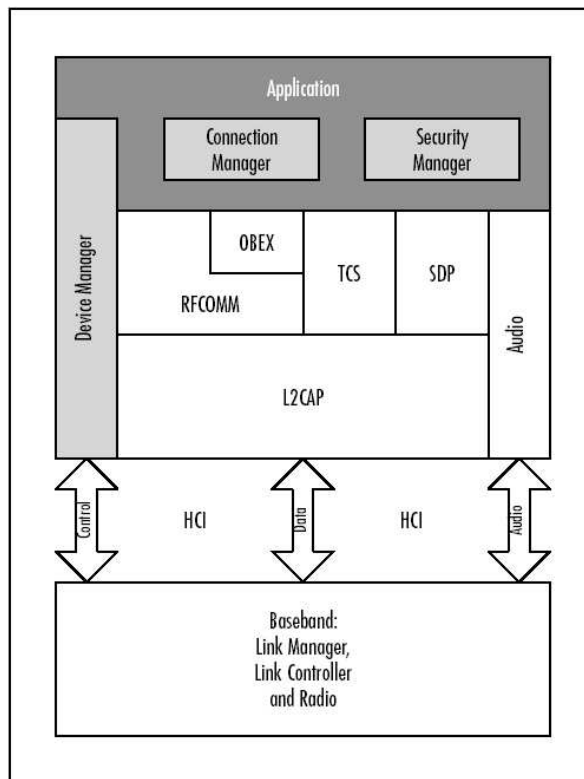

Fig. 3. Bluetooth Stack Layers

Profile	Lower Layers	L2CAP	SDP	RFCOMM	PPP	OBEX	TCS-Bin
Service Discovery Application	X	X	X				
Cordless Telephony	X	X	X				X
Intercom	X	X	X				X
Serial Port	X	X	X	X			
Headset	X	X	X	X			
Dial-up Networking	X	X	X	X			
FAX	X	X	X	X			
LAN Access	X	X	X	X	X		
Generic Object Exchange	X	X	X			X	
Object Push	X	X	X			X	
File Transfer	X	X	X				
Synchronization	X	X	X			X	

Fig. 4. Layers involved in each profile

and L2CAP allocates a CID. There may be several open channels carrying the same PSM data. Each Bluetooth defined layer above L2CAP has its own PSM:

- SDP - 0x0001
- RFCOMM - 0x0003
- TCS-Bin - 0x0005 or 0x0007

RFCOMM (a name coming from an Radio Frequency-oriented emulation of the serial COM ports on a PC) emulates 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 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.

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.

A Bluetooth system is a multitasking system. Therefore, there are many different tasks to take care of: Link Management messages must be processed; incoming data must be dealt with as it arrives; outgoing data has to be sent to the baseband and radio; if there is a separate host communications through the HCI this must be addressed; all this and more must be handled simultaneously. In order to deal with all tasks simultaneously, any multitask system needs a scheduler. One possible solution to include a scheduler in a system is using an operating system. 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 [2], and it includes stable HCI, L2CAP, and RFCOMM drivers, as well as user-space SDP applications.

III. Measurement Environment

We installed in two systems 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 CPU 2.80 GHz running Linux 2.4.22, with the Xilinx Platform Studio Release 7.1.1 and iMPACT H.40 Release 7.1.02i, part of Xilinx Embedded Development Kit 7.1.[3].

• The Xilinx ML310 Embedded Development Platform [4] with a Virtex-II Pro FPGA and two Power PC 405 CPU 300 MHz, one of them running Linux MontaVista 2.4.24 [5]. The second PPC is not used in these tests.

One of the most useful applications (profiles) that can be executed over Bluetooth is 'file transfer'. This profile will imply the use of the lower layers, L2CAP and SDP. In our case, as we are using the Bluetooth USB adapter from Conceptronic [6], 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 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.

IV. Bluetooth Profiling Results

In Figure 5, the first 45 rows of the profiling information of one of these trials, sorted by number of ticks (N.Ticks), are depicted. The N.Ticks is a value returned by the profiler and is a measure of how many times a particular function was running on the processor when the system was interrupted to retrieve the profiler information. Figure 6 depicts the first 45 rows sorted by Normalized Load, which is also returned by the profiler. It is calculated by dividing the number of ticks a function has recorded by the length of the address space the function occupies in memory [7]. Therefore, it is safe to assume that functions rate high in both lists (Figures 5 and 6) occupy the processor the most.

From Figures 5 and Figure 6, we can easily identify six Bluetooth functions: *hci_usb_rx_complete*, *l2cap_recv_acldata*, *hci_usb_rx_submit*, *hci_send_to_sock*, *hci_rx_task* and *hci_acldata_packet*. Figure 7 depicts the statistical Mean and the Standard Deviation (STD) of the number of ticks of these Bluetooth functions. For each function it is also presented the convergence. This is defined as follows:"The experiments converged to an x% confidence interval width at y% confidence". This means that the results of a confidence interval estimated based on a Students's T-test has a width of less than x% of the Mean with y% level of confidence [8]. The obtained convergence shows how meaningful the results of these 20 trials are.

V. Global Profiling Results

From an analysis of both Figures 5 and 6, it is clear that the functions called are a mixture of OS, TCP/IP and Bluetooth functions. We can also identify that in the majority of the 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 intensive ones, however they are not 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 different each time the same

N. Ticks:	Function Name:	Normalized Load:
67513	__sti_end	2109.7812
2252	__sti	28.1500
1245	__save_flags_ptr_end	12.4500
205	memcpy	1.3141
182	hc_interrupt	0.3730
163	speedo_interrupt	0.2735
140	trident_interrupt	0.5556
92	XSysAce_RegRead32	1.0952
83	dl_done_list	0.2096
81	sohci_submit_urb	0.0671
75	__copy_tofrom_user	0.1330
74	hci_usb_rx_complete	0.1729
54	dl_transfer_length	0.1688
52	sys_select	0.0478
51	tcp_recvmsg	0.0236
51	skb_under_panic	0.0823
50	hci_usb_rx_submit	0.1025
49	__save_flags_ptr	0.6125
48	l2cap_recv_acldata	0.0619
45	hci_send_to_sock	0.0787
45	fget	0.6618
44	hci_rx_task	0.1310
44	do_select	0.0827
44	__kmem_cache_alloc	0.1111
42	sys_read	0.1265
39	sys_write	0.1175
39	hci_acldata_packet	0.1147
39	__kfree_skb	0.1005
38	XSysAce_RegWrite32	0.7308
37	tcp_v4_rcv	0.0238
35	power_save	0.3017
34	usb_submit_urb	0.3400
33	tcp_rcv_established	0.0154
32	bnep_rx_frame	0.0289
31	ppc_irq_dispatch_handler	0.0605
30	DoSyscall	0.3261
29	normal_poll	0.0780
29	kmalloc	0.3625
27	memset	0.2935
26	tcp_poll	0.0619
26	tasklet_action	0.1182
26	hci_recv_frame	0.1032
25	sohci_return_urb	0.0625
25	schedule	0.0235
1059	Others	
74383	Total	0.0424

Fig. 5. The first 45 rows of the profiler sorted by number of ticks

Normalized Load:	Function Name:	N. Ticks:
2109.7812	__sti_end	67513
28.1500	__sti	2252
12.4500	__save_flags_ptr_end	1245
1.3141	memcpy	205
1.0952	XSysAce_RegRead32	92
0.7308	XSysAce_RegWrite32	38
0.6618	fget	45
0.6125	__save_flags_ptr	49
0.5556	trident_interrupt	140
0.3730	hc_interrupt	182
0.3625	kmalloc	29
0.3400	usb_submit_urb	34
0.3261	DoSyscall	30
0.3017	power_save	35
0.2935	memset	27
0.2735	speedo_interrupt	163
0.2096	dl_done_list	83
0.1729	hci_usb_rx_complete	74
0.1688	dl_transfer_length	54
0.1330	__copy_tofrom_user	75
0.1310	hci_rx_task	44
0.1265	sys_read	42
0.1182	tasklet_action	26
0.1175	sys_write	39
0.1147	hci_acldata_packet	39
0.1111	__kmem_cache_alloc	44
0.1032	hci_recv_frame	26
0.1025	hci_usb_rx_submit	50
0.1005	__kfree_skb	39
0.0827	do_select	44
0.0823	skb_under_panic	51
0.0787	hci_send_to_sock	45
0.0780	normal_poll	29
0.0671	sohci_submit_urb	81
0.0625	sohci_return_urb	25
0.0619	tcp_poll	26
0.0619	l2cap_recv_acldata	48
0.0605	ppc_irq_dispatch_handler	31
0.0478	sys_select	52
0.0289	bnep_rx_frame	32
0.0238	tcp_v4_rcv	37
0.0236	tcp_recvmsg	51
0.0235	schedule	25
0.0154	tcp_rcv_established	33
	Others	1059
0.0424	Total	74383

Fig. 6. The first 45 rows of the profiler sorted by Normalized Load

interrupt is issued. As we do not know in advance what will be the executed code, we cannot implement it on a reconfigurable processor.

What remains is the *memcpy* function, which is responsible for copying data of size *count* from address *src* to *dest* (the C code is presented next).

```
/**
 * This function is defined in
 * linux-2.4/lib/string.c
 *
 * Copy one area of memory to another
 * @dest: Where to copy to
 * @src: Where to copy from
 * @count: The size of the area.
 *
 * You should not use this function to
 * access IO  space, use memcpy_toio()
 * or memcpy_fromio() instead.
**/
void *memcpy(void *dest,const void *src,
             size_t count)
{
  char *tmp=(char *)dest, *s=(char *)src;

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

  return dest;
}
```

This function is called by several Bluetooth stack functions but also by other TCP/IP and system functions. Figure 8 is a simplified scheme that depicts some of the functions that call *memcpy*.

Transferring a file of 50 MB, implies that each frame

received has to be transferred from the receiver buffer to some other location so it could be accessed after completion of the transfer. Then, *memcpy* is called many times and it occupies the processor for a period of time that is depended of the size of data it is moving, each time it is called. As such, *memcpy* is the most likely candidate function to be implemented on the reconfigurable processor.

VI. Conclusions

We have observed a trend of 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 achieved 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 six functions as the most intensive 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 intensive function is *memcpy*. As such, this is the function proposed to be implemented on a reconfigurable processor.

References

[1] "Bluetooth Specification v1.2." `http://www.bluetooth.org/spec/`.

[2] "BlueZ." `http://www.bluez.com`.

[3] "Xilinx Embedded Deveolpment Kit." `http://www.xilinx.com/edk`.

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

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

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

[7] M. Wong, "Stressing linux with real-world workloads," in *Proceedings of the Linux Symposium*, 2003.

[8] R. Bryant, B. Hartner, Q. He, and G. Venkitachalam, "Smp scalability comparison of linux kernels 2.2.14 and 2.3.99," in *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

[9] D. Kammer, G. McNutt, B. Senese, and J. Bray, *Bluetooth Application Developer's Guide: The Short Range Interconnect Solution*. Syngress Publishing, Inc., 2002.
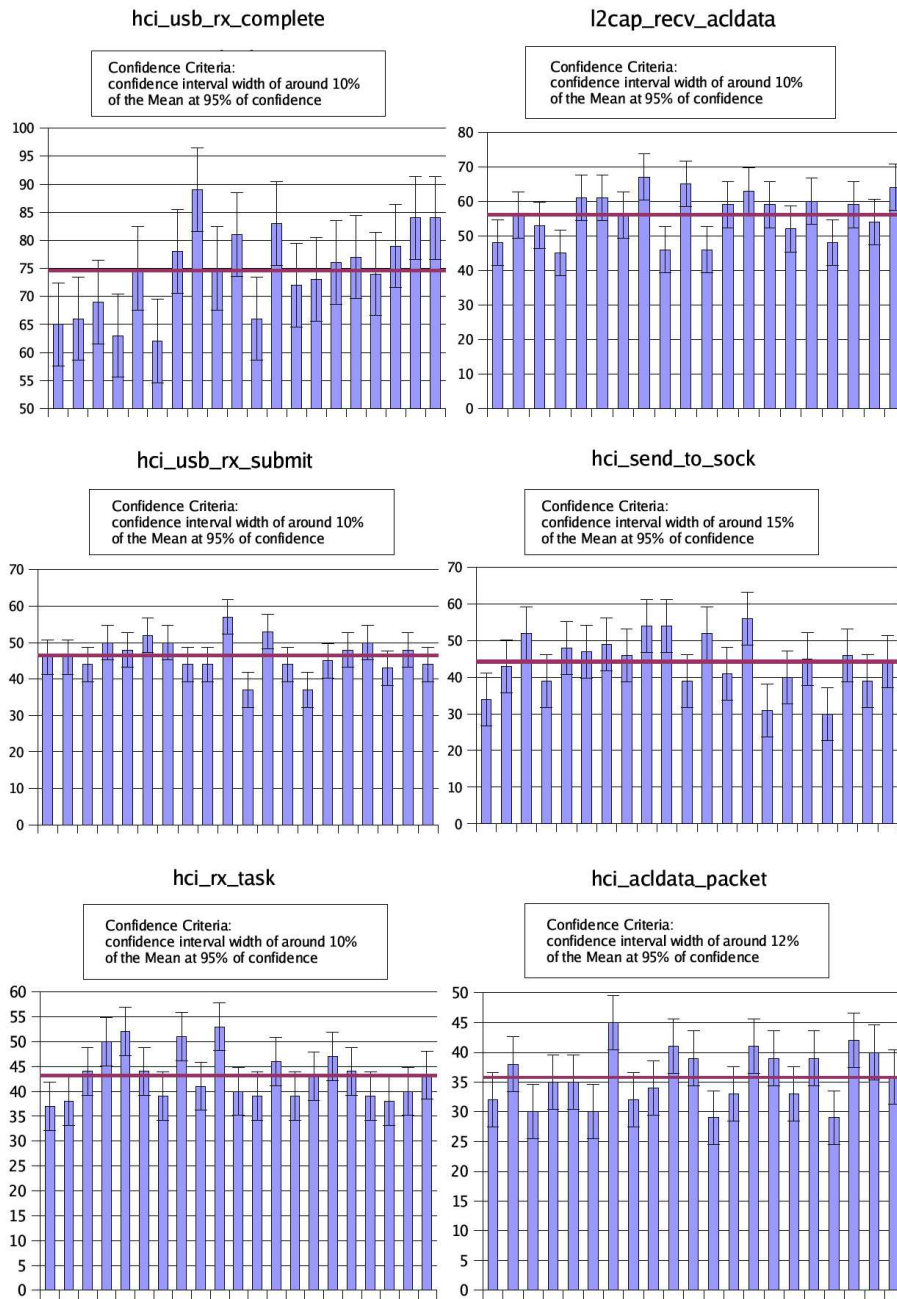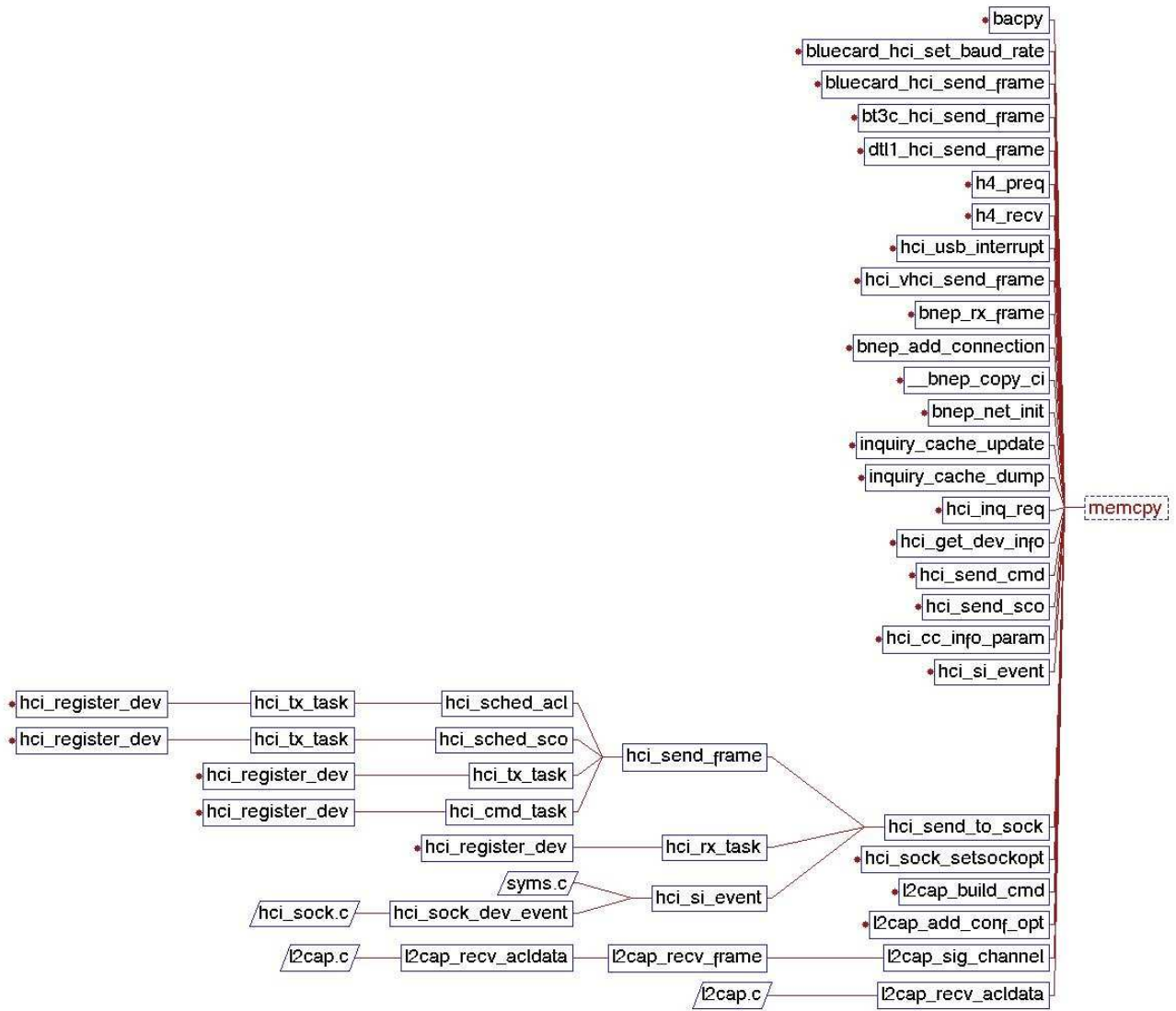
Fig. 7. Mean and Standard Deviation of the top six Bluetooth functions

Fig. 8. Functions called by *memcpy*