System Architecture for Software Peripherals

Siddharth Choudhuri
Center for Embedded Computer Systems
University of California
Irvine, CA 92617, USA
e-mail: sid@cecs.uci.edu

Tony Givargis
Center for Embedded Computer Systems
University of California
Irvine, CA 92617, USA
e-mail givargis@uci.edu

Abstract— Software peripherals [1] have been proposed as a design alternative to traditional peripherals. We propose a software architecture, design methodology and scheduling scheme for implementing software peripherals on general purpose processors, with fast context switch and high resolution timers. Our design flow automatically generates code for scheduling software peripherals. We demonstrate the feasibility of our proposed work by experimenting with a set of five software peripherals scheduled to execute on a MIPS processor. Our performance evaluations show that the performance impact of the software peripherals on user-level tasks is minimal (i.e., 10.11% on a 100 MHz processor) – strongly suggesting that with the right architecture, software peripherals can be efficiently accomodated in typical embedded applications.

I. Introduction

The complexity of embedded systems is on the rise. This is driven by several factors such as, the possibility of integrating multi functional components into a single system i.e., System-on-Chip (SoC) [2], the demand from consumer and other industry segments to have end products that serve more than one purpose (for example cell phones morphing into personal digital assistants, communication device, gaming device). The complexity of systems coupled with increasing time pressure to launch newer products has lead to the design of embedded systems that are highly tuned for specific target applications. Such embedded systems are based on a tightly coupled processor-peripheral platform.

Ideally, a processor-peripheral platform is designed for flexibility, cost and performance. Flexibility leads to design reuse and shorter time to market when developing new products derived out of existing, well known and tested platforms. Flexibility also reduces non-recurring engineering (NRE) costs. A single platform consisting of processor-peripheral helps in cost savings during manufacturing. Performance is required to run complex software applications on these platforms.

Software peripherals [1, 3] have been proposed to meet the challenges of flexibility and cost in processor-peripheral platforms, while still meeting performance requirements. Software peripherals emulate the implementation of peripherals in software (processor) with minimal hardware support, if any. The software implementation produces bit patterns using the processor I/O pins such that the pattern generated has the same frequency as the hardware implementation of the peripheral. This leads to greater flexibility, lower cost (reduced hardware) and easier functional upgrades. System design translates into a processor with minimal hardware support that can be configured to have a combination of peripherals, each running on the processor as a software task. With growing speeds of embedded processors, the performance requirements can be met while accommodating software peripheral tasks.

In this paper, we present software architecture and scheduling for software peripherals. To our knowledge, the proposed approach is a first step towards implementation of software peripherals on general purpose processor architecture with support for fast context switch and high resolution timer. Further, we propose a design flow to automatically generate a schedule layout based on interrupts followed by experimental results from simulation.

The remainder of this paper is organized as follows: The next section discusses related work and our contributions. The details of system architecture is described in section 3. The results and analysis of results is provided in section 4 followed by summary and conclusion.

II. Related Work

The idea of software peripherals was proposed in [1, 4]. The focus in [1, 4] was on proposing the idea of software peripherals, its usefulness and estimating the processor overhead imposed by software peripherals. This work used a simplified model of peripherals to calculate the processor utilization for each peripheral in isolation. However, to realize a feasible system based on the idea of software peripherals would require scheduling multiple peripherals as software tasks and a detailed model for each peripheral. In our work, we address the following issues (i) a methodology and design flow to implement software peripherals (ii) scheduling scheme that outputs a schedule layout to be invoked with the help of a high resolution timer (interrupt) (iii) a detailed model of peripheral that implements the functionality of peripheral as a software task (iv) system overhead from simulation.

Ubicom [5, 6, 3] has an implementation of software peripherals targeted for networking devices. Ubicom processors provide an almost zero context switch overhead resulting in an extremely efficient implementation of software peripherals. Every fetch-decode cycle is preceeded by a software peripheral task instruction. In our work, we explore the possiblities of realizing software peripherals on conventional processor architectures that do not have a datapath specifically designed for software peripherals.

Triscend provides configurable processor based on standard microcontroller or microprocessor architectures. Users can add desired peripherals using a GUI based tool (*FastChip configuration*). Once designed, the peripherals are implemented as cores. Thus, the flexibility is only at the design phase and design reuse is not an option.

There has been a growing trend of implementing conventional peripherals in software and reconfigurable hardware in order to attain greater flexibility in design and reuse. Some of the examples are software modem, software defined radio and several other reconfigurable architectures [7, 8, 9, 10]. However, implementing peripherals in reconfigurable hardware is orthogonal to our work, though the goal is to have flexibility and reusability in design. Our goal is to have minimal hardware, if any, and push the peripheral functionality to software (processor).

III. Technical Approach

The implementation of peripherals in software is transparent to the user applications. Writing data to a peripheral at the application level translates into a write request made by the device driver to the software peripheral implementation. The software peripheral implementation translates the data it gets from device driver into a sequence of bit patterns. This bit pattern is sent to the physical I/O of the processor at a certain rate, resulting in a sequence of 1s and 0s at the output pins of the processor that excatly mimics the peripheral functionality in *frequency* and *sequence*. Similarly, the software peripheral is also responsible for reading I/O pins at a specific, pre-determined frequency and notifying the higher level tasks of any input events. Software peripherals run in the context of an OS. This is due to the fact that the implementation requires low level, privileged access to pin I/O routines and interrupt handlers, details of which are provided in the next section.

A. System Architecture

Figure 1 shows our proposed architecture for software implementation of peripherals. In general, a peripheral's operation can be divided into: (i) *Computation task* that implements the functionality and (ii) *Communication task* that deals with bit I/O at the hardware interface.

In our proposed architecture, the *High Level Peripheral (HLP) Task* is responsible for the computation part of a given peripheral. It contains the implementation of the peripheral (which in a conventional architecture would have been in a dedicated hardware). In case of a write request, the output of the HLP task is a sequence of bit pattern written to peripheral memory buffer. During a read request to a peripheral, the HLP task reads a sequence of bit pattern from the peripheral memory buffer, processes the bit pattern and returns the result to the device driver. Note that, the driver gets the same data that would have been returned by a hardware peripheral.

The *Peripheral Memory Buffer* is a shared memory region between the HLP task and the Low Level Peripheral (LLP) task. It is a pool of buffers wherein each instance of a peripheral has a read buffer and a write buffer, depending on the peripheral functionality [1]. The content of buffers are bit patterns with start and end pointers maintained by the HLP and LLP tasks.

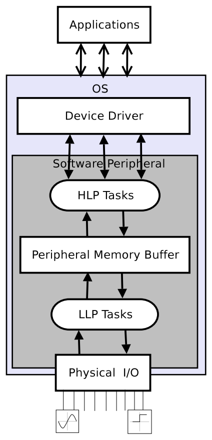


Fig. 1. Software Peripheral Architecture

Low Level Peripheral (LLP) Task : The LLP task is responsible for pin wiggling and reading the pin inputs (i.e., bit input-output). In case of a peripheral write operations, the LLP task reads the bit pattern from peripheral memory buffer and outputs it to the processor I/O pins. Similarly, during a read operation, the LLP reads the status of processor I/O pins and appends it to the appropriate buffer. The LLP task has stringent requirements in terms of its periodicity. Not invoking LLP task at right time(s) can lead to incorrect outputs and incorrect interpretation of data read by the peripheral. This requirement makes the LLP task a hard realtime task. Since the peripheral characteristics are known apriori, LLP task is statically scheduled. Due to the nature of the LLP task, it has to run on a high priority interrupt timer. The LLP task is invoked atleast as frequently as the fastest peripheral.

The rationale behind dividing the peripheral functionality into HLP and LLP task is that a peripheral task needs to be invoked atleast as frequently as the highest frequency peripheral. However, most of such invocations would be to read/write bit patterns (communication) and a few that implement the peripheral functionality (computation). Given the high rate of invocation, a small interrupt service routine would suffice to do the pin I/O. On the other hand, a software task with peripheral implementation will need to be invoked less often but such a task would be computation intensive. Note that the HLP and LLP tasks have a producer-consumer relationship depending on a write or read operation being performed.

In order to implement the LLP task, certain architecture support is required. An extremely high resolution timer interrupt is required to invoke the LLP task at high frequencies. Also, the context-switch overhead needs to be minimal or close to zero. An almost zero context switch is possible by using processor architectures that support register banking. One set of register bank can be dedicated to execute LLP task. Thus context switch overhead amounts to switching from one bank to another. Register banks is not a new concept and has been used in architectures like 8051 and ARC processor [11, 12]. Architec-

[1]Eg: keypad only requires a read buffer, whereas serial port requires both read and write buffer

ture support is also required in terms of A/D and D/A converters to realize an implementation of peripherals that have analog interface.

B. Scheduling LLP Tasks

Each peripheral's low level bit input-output is modeled as a realtime task $< p,\ e,\ d >$ with periodicity p, execution time e and deadline d. The periodicity is the rate at which the peripheral operates. Execution time is the the number of instructions in the LLP task. Deadline is derived from data-sheet specification, depending on how much delay is tolerable. In most cases, deadline is equal to the execution time. However, for peripherals where a certain amount of delay is tolerable, deadline is greater than execution time.

A sample schedule for software peripherals is shown in Figure 2. Here, two peripherals with low level task characteristics, P_1 = $< 9,\ 0.5,\ 0.5 >$ and P_2 = $< 11,\ 0.5,\ 0.5 >$ are shown as examples. These two tasks are statically scheduled. Note that (a) The LLP tasks run with the highest priority, (b) The execution time of LLP task is much smaller than any other tasks. (c) $ResponseTime$ is defined with respect to the applications. An application, during its course of executtion, might get pre-empted due to an LLP task interrupt. The response time is defined as the duration of time that an application is in pre-empted state due to invocation of an LLP task. (d) $Slack$ is also defined with respect to the applications. It is the amount of processor time an application gets between two consecutive invocations of LLP task interrupts.

The inputs to the LLP task scheduler are a set of N tasks $\{T_1, T_2, \cdots, T_N\}$ corresponding to N peripherals and the context switch overhead of the given processor, CS. The output of the algorithm is a schedule layout if successful. The scheduling is based on a scheme of iterating through every combination of peripherals till all combinations are exhausted or a valid schedule is found.

Step 1 tests for schedulability based on [13] followed by step 2 that calculates the hyperperiod of the task set. Note that the length of schedule layout is equal to the hyperperiod i.e., $|L| = H$. Step 3 explores every possible combination of the task set until either all combinations are exhausted or a valid schedule is found (step 17). Step 4, schedules the first task. This is followed by scheduling rest of the tasks in steps 5 through 16. Each task T_i has H/p_i instances that need to be scheduled in the layout. Steps 7 though 11 try to schedule each of this instance. If an instance of task cannot be scheduled at a certain time (because there exists some other task whose instance is already scheduled), the deadline information is used to determine if a slack is tolerable. This is done in step 9. If any instance of task cannot be scheduled, the existing layout is cleared (step 13) and the next combination of tasks is tried starting at step 3.

```
Input  Tasks $T = T_1, T_2, ..., T_N$, Context Switch Overhead $CS$
Output  Schedule layout if successful
1.  Test schedulability of $T$ using $U > N(2^{1/N} - 1)$
2.  Calculate Hyperperiod $H = LCM(T_1, T_2, ..., T_N)$
3.  For each combination of task set $T_c = permute(T)$
4.       Schedule $T_1 \in T_c$ at $t = 0$, add $T_i$ to layout $L[0]$
5.       For each task $T_i = 2$ to $N$
6.            For $j = 1$ to $(H/p_i)$
7.                 Try schedule $T_i$ at times $t = j \times p_i$
                   with execution time $e_i$
8.                 If step 7. fails
9.                    Try step 7. with available slack
                      $[0,\ (d_i - e_i)]$
10.                        If step 9. fails for all
                           values $[0,\ (d_i - e_i)]$ break
11.           End For
12.           If $T_i$ is not scheduled
13.              Invalidate $L$, break
15.           Add $T_i$ to $L[t]$
16.      End For
17.      If task set $T = \{T_1, T_2, \cdots T_n\} \in L$,
            $scheduled \Leftarrow true$  break
18.  End For
19.  If $scheduled = true$
20.      For $i = 0$ to $(|L| - 1)$
21.           If $((L[i+1] - L[i]) \leq CS)$
              Merge $L[i], L[i+1]$
23.      End For
24.  return $scheduled$
```

If a layout is possible and determined, the next step is to take the context switch overhead into account and coalesce any two neighboring tasks such that the slack between them is less than the context switch overhead. This slack can be filled with no-ops, instead of doing a context switch and missing the next task deadline.

The complexity of this algorithm is $O\left((N!) \times N \times H/p_{min}\right)$, where p_{min} is the task with minimum periodicity i.e., (H/p_{min}) is maximized. Though the complexity is exponential, it is to be noted that the schedule is computed offline and the number of peripherals, N, does not reach a high value. Our worst case running time is less than 10 minutes with $N = 5$.

C. LLP Implementation

The implementation and memory layout requirements of LLP task is described in this section. Figure 3 shows a snapshot of the memory layout of LLP task and peripheral memory buffer. The figure depicts an implementation of four peripherals. The hyperperiod of tasks is 10 units of time. The arrays `schedule` and `T` are generated by the scheduling algorithm described previously. The `schedule` array is of length hyperperiod and consists of the sequence in which LLP tasks are invoked. For example, in this case, peripheral P_1 is invoked five times in the given layout. The array T is used to invoke the timer interrupt. For example at time $t7 = T[7]$ the LLP task corresponding to peripheral $P_3 = schedule[7]$ is invoked.

The peripheral buffer consists of buffer for each peripheral as described in the previous section. The write buffer for serial port peripheral is shown in the figure as array `spi_wr_buf`. This buffer is shared between the HLP task and LLP task. For example, in case of serial port, the HLP task calculates parity and adds start, stop bits to every byte and places it in `spi_wr_buf` array. This is followed by moving the `end` pointer. Note that (a) The peripherals are circular buffer and therefore, the `start` and `end` pointers roll back. (b) There does not arise a case in which `end` overwrites the `start` i.e., data is never produced by the HLP task at a rate higher than LLP task can consume (and vice-versa). (c) The process of incrementing `end` pointer has to be an atomic operation to avoid the case of HLP task being interrupted by the same LLP task

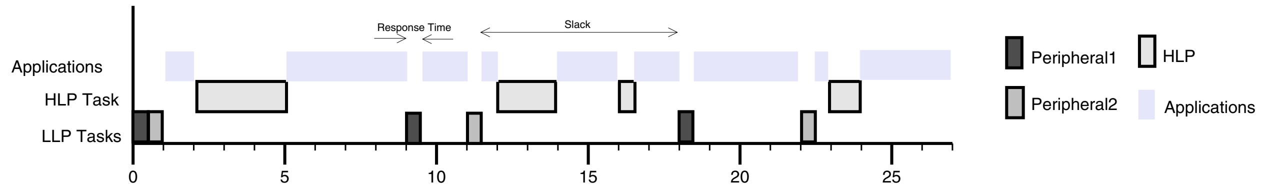


Fig. 2. Three Level Tasks Schedule

and having `start` and `end` pointers pointing to the same location. A similar requirement does not hold for LLP task because the LLP task runs in the context of a high priority interrupt handler that is not pre-empted. (e) There can be a case when a peripheral is inactive, in this case the LLP task sends the default bit to the physical I/O. For example in case of serial port the default behavior (i.e., when no bits are shipped) is a high state. This condition is easily identifiable if one observes that a peripheral is in default state when `start` and `end` pointers point to the same location.

The code for LLP task is implemented as an interrupt handler (`intr_handle()` function) having case statements. Each case statement has code for a peripheral LLP. For example, in figure 3, there are four case statements corresponding to the four peripherals. The interrupt handler is invoked by a high priority timer interrupt at times encapsulated by the timer array T. Note that before exiting the handler, the timer is reset to the next invocation (`set_timer()` function), which is $T[i] - T[i-1]$.

IV. Experimental Results

We have considered five peripherals for their software implementation. The peripherals vary in their HLP execution cycles, operating frequency and the number of pins required for I/O. Table I lists the peripherals. Specifically, *Serial Port* is a UART implementation with 19200 baud rate and 8E1 configuration. *Keypad* is a software implementation of a 4×4 keypad controller having row-column decoder. *Timer* is a software implementation of a timer-interrupt generator with a configurable rate. *PWM* is a software implementation of a configurable pulse width modulator. *Modem* is a software implementation of the V.34 modulation protocol adopted from [14]. Columns 2 and 3 of Table I are execution cycles of the HLP and LLP task derived by profiling the code on a simulator. The last column is the number of processor GPIO (General Purpose I/O) pins required.

TABLE I
Emulated Peripherals

Peripheral	HLP Cycles	LLP Cycles	Frequency	Pins
Serial Port	364	64	19200 baud	2
Keypad	16	29	1 KHz	8
Timer	10	31	10 KHz	1
PWM	90	34	10 KHz	1
V.34 Modem	7660	32	33600 bps	2

A. Experimental Setup

Figure 4 depicts the experimental setup and design flow. *Peripheral Models* in figure 4 capture peripheral properties such as the operating frequency, the LLP execution time, and the deadline for each peripheral. The periodicity p of the peripheral is determined from the data sheet. Profiling an LLP task determines the execution cycles e. The *Task Generator* is a script that takes as inputs the CPU frequency and the peripheral models. It generates a task set file containing the LLP task tuples $(p_i,\ e_i,\ d_i)$. A sample task file for processor speed of 100 MHz is given in Table II.

TABLE II
Task File - 100 MHz

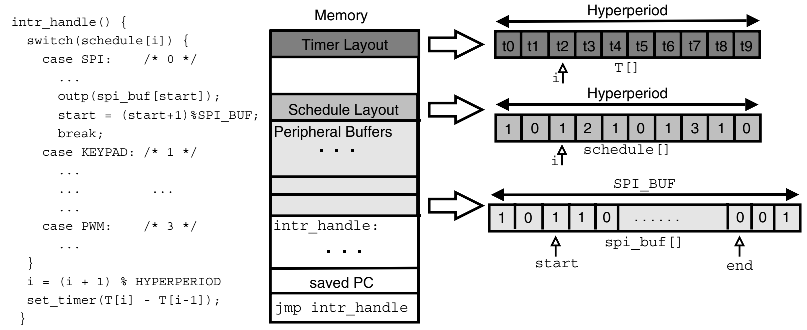
Task	Period	Exec Cycles	Deadline
Serial Port	5208	64	64
V.34 Modem	2976	32	32
Keypad	100000	29	57
Timer	10000	31	31
PWM	10000	34	34

The inputs to the *Schedule Generator* are the task sets, the CPU frequency and the context switch overhead. The schedule generator implements the schedule layout algorithm described in the previous section. The schedule layout is captured in a file `llp.h`. The time instances of when each task is invoked is captured in the array `T` and which task to invoke in array `schedule`. In addition to the layout, each peripheral is assigned a task-id. Since the order of execution of LLP tasks is not known until a layout is generated, task-id serves the purpose of resolving the order, thereby executing peripheral specific code in the `case` statement of the interrupt handler (Figure 3, `intr_handle` function). Further, the inputs to the compiler are the application source code, the HLP task source that has the peripheral implementation, the interrupt handler source that implements the LLP task, and the generated layout header file. The binary thus generated, directly runs on our MIPS instruction set simulator.

B. Results

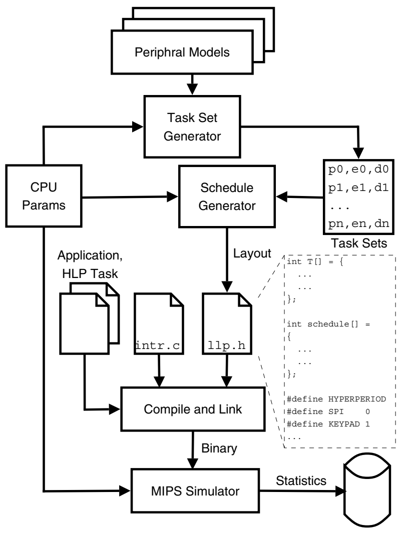
We now present our experimental results. We have considered four different CPU frequencies (100, 150, 200 and 250MHz), representative of embedded processors.

The first set of results in Table III depicts overhead percentage of the LLP and HLP tasks. For a given CPU frequency of c, execution time e, and peripheral operating frequency f the



```
intr_handle() {
  switch(schedule[i]) {
    case SPI:    /* 0 */
      ...
      outp(spi_buf[start]);
      start = (start+1)%SPI_BUF;
      break;
    case KEYPAD: /* 1 */
      ...
      ...          ...
      ...
    case PWM:    /* 3 */
      ...
  }
  i = (i + 1) % HYPERPERIOD
  set_timer(T[i] - T[i-1]);
 }
```

Fig. 3. Memory Layout



```
int T[] = {
  ...
  ...
};

int schedule[] =
{
  ...
  ...
};

#define HYPERPERIOD
#define SPI    0
#define KEYPAD 1
...
```

Fig. 4. Experimental Setup

TABLE III
PROCESSOR OVERHEAD

Peripheral	**100MHz**			**150MHz**			**200MHz**			**250MHz**		
	LLP	HLP	Total	LLP	HLP	Total	LLP	HLP	Total	LLP	HLP	Total
Serial Port	1.22	0.70	1.92	0.82	0.46	1.28	0.60	0.34	0.94	0.48	0.35	0.83
Modem	1.07	6.43	7.50	0.71	4.29	5.00	0.53	3.22	3.75	0.43	2.57	3.00
Keypad	0.02	0.02	0.04	0.01	0.01	0.02	0.01	0.01	0.02	0.01	0.01	0.02
Timer	0.31	0.10	0.41	0.20	0.07	0.27	0.15	0.05	0.20	0.12	0.04	0.16
PWM	0.34	0.11	0.45	0.17	0.08	0.25	0.22	0.06	0.28	0.13	0.05	0.18

overhead of the LLP task is equal to $((f \times e)/c) \times 100$ percent. The HLP task overhead is calculated based on a worst case scenario i.e., assuming a peripheral is never idle. For example, in case of serial port, the HLP task is assumed to be always transmitting and receiving data. Note that, the rate at which HLP task is invoked is less frequent than that of the LLP task invocation. For example, in case of the serial port, one invocation of the HLP task results in 10 bits of data being processed, 8 bits of data per byte to be transmitted along with one stop bit, and one bit for even parity. Thus the HLP task is invoked $f_{serial}/10$ times compared to the serial LLP task's periodicity of f_{serial}. Also, the values are computed based on Table I, which takes into account the execution cycles due to function call overhead (saving and restoring registers), in addition to the actual cycles due to processing data. It can be seen from Table III that even in case of a software modem which requires a large number of cycles to implement the functionality in software, the overhead on processor is minimal.

Our next set of results are based on a system configuration consisting of all the peripherals mentioned in Table I connected to a processor. The results are based on calculating the hyperperiod and running the simulation for one hyperperiod. The schedule for the five peripherals is computed based on flow described in Figure 4. Table IV depicts the results. In order to come up with a valid schedule, having a reasonable hyperperiod, the periodicity of two peripherals (serial port and modem) had to be adjusted. This resulted in the certain amount of error from the actual periodicity. However, peripherals can tolerate certain amount of jitter in periodicity without actually giving an incorrect result. The larger of the two errors introduced resulted in a deviation of 413 nsec, which was a tolerable jitter. The total cycles depicted in column 11 is the amount of time (in clock cycles) the simulation runs. Note that it is close to the hyperperiod in each case, but not exactly same as the hyperperiod. This is due to the fact that there is some amount of cycles spent in starting the simulation and invoking the LLP task. Also, as in previous result, the assumption here is that the peripherals are always active which implies that the HLP task runs for the maximum possible number of times. Note that the worst case utilization is 10.1% in case of a 100 MHz processor. Column 10 (Intr.) is the number of times the low level interrupt is invoked to run the LLP task. Note that this value could be same for different processor speeds i.e., if the same schedule layout is made for two different processor frequencies, the number of times the LLP task is invoked will be the same. In this particular example, the schedule layout in case of 100 MHz, 200 MHz, and 250 MHz is the same i.e., the same schedule is generated for the above three processor frequencies. The hyperperiod have different values as the units are cycles and not time.

The response time is expressed as a set $R = \{(r_1, n_1), (r_2, n_2), \cdots\}$, where r_i is the response time and n_i is the number of such instances. Similar to the case of low level interrupts (Intr. Column 11), the response time R is also same for a given schedule layout, independent of processor frequncy. The response time R for processor frequencies of 100 MHz, 200 MHz and 250 MHz is given by

TABLE IV
PROCESSOR UTILIZATION

CPU (MHz)	**Hyper-period**	**Error %** Serial	Modem	Keypad	PWM	Timer	**HLP (%)**	**LLP (%)**	**Intr.**	**Total cycles**
100	3900000	-0.15	0.80	0.0	0.0	0.0	7.13	2.98	2869	3999900
150	21450000	-0.10	-1.38	0.0	0.0	0.0	4.88	2.08	10628	21834904
200	7800000	-0.15	0.80	0.0	0.0	0.0	3.60	1.51	2869	7899198
250	9750000	-0.15	-0.80	0.0	0.0	0.0	2.89	1.21	2869	9847698

$\{(32, 1238), (64, 700), (65, 351), (94, 26), (126, 12), (190, 1)\}$. Similarly the response time for processor frequency of 150 MHz is given by $\{(32, 4738), (62, 1287), (64, 2625), (91, 130), (96, 124), (123, 12), (187, 1)\}$. The best case response time is 32 cycles and the worst case response time is 187 cycles (for 150 MHz) and 190 cycles (for other processor frequencies). Note that there are only a few instances where response time is large and a large number of instances where the response time is small. Thus, scheduling LLP tasks does not have a considerable impact on the response time of higher level tasks. Theoretically, the response time for higher level tasks in case of N peripherals (assuming zero context switch overhead) is bounded by $[\min_{i=1}^{N} e_i, \ \sum_{i=1}^{N} e_i]$, where e_i is the LLP task execution time of peripheral P_i. The lower limit is the case when an LLP task with least execution time is executed and there is no other LLP task to be executed thereafter. The upper limit case is when all LLP tasks are scheduled one after the other. Figure 5 shows the distribution of slack for the four processor configurations. The horizontal axis depicts the amount of slack in cycles and the vertical axis depicts the number of instances of such slack available. The extremely large number of cycles available in each case (the spike observed in each case), is due to the large variation in the periodicity exhibited by the peripherals. This is evident from the Table II. A case where the variation in periodicity is not large enough exhibits a slack distribution that is flat compared to Figure 5. However, this is not shown in this paper due to lack of space. It can be seen from figure 5 that non LLP tasks do get sufficient amount of time to execute without being interrupted by LLP tasks.

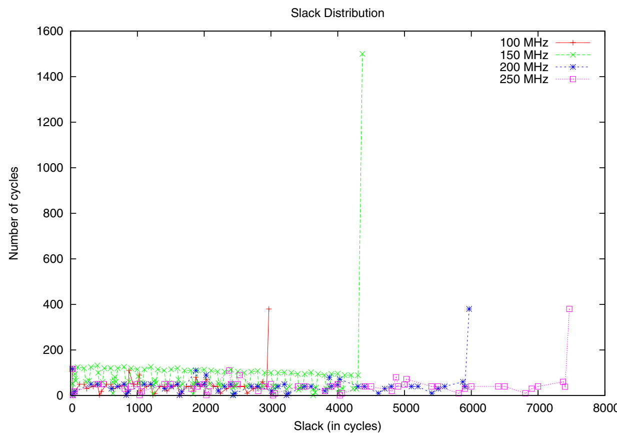


Fig. 5. Slack Distribution

In conclusion, we proposed a design flow and software architecture for realizing software peripherals on processors with high resolution timers and fast context switch support. Our results show that with the right hardware architecture support, it is possible to schedule software peripherals with minimal impact on other high level applications. However, the complexity of a peripheral might be a limiting factor with traditional processor architecture support.

With the emergence of MPSoCs and NoCs, in future, it may be possible to dedicate on chip cores to perform specific peripheral functionality. We have not considered the power and energy considerations of having software peripherals. While the processor power consumption might increase due to HLP and LLP tasks, not having peripheral hardware may result in energy savings. This is one of the areas to be looked in future.

REFERENCES

[1] D. Lioupis, A. Papagiannis, and D. Psihogiou. A systematic approach to software peripherals for embedded systems. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 140–145, New York, NY, USA, 2001. ACM Press.

[2] P.Cummings. The TI OMAP Platform Approach to SoC. In *Winning the SoC Revolution*. Kluwer Academic Publishers, 2003.

[3] Jim Turley. Soft peripherals. In *Embedded Systems Programming*, May 2003.

[4] D. Lioupis, A. Papagiannis, and M. Stefanidakis D. Psihogiou. Software peripherals - requirements and constraints for real-time embedded systems. In *IEEE Real-Time Embedded Systems Workshop*, 2001.

[5] Whitepaper. The Ubicom IP2022 Multiprotocol processor.

[6] Whitepaper. The Ubicom IP2023 wireless network processor.

[7] Michael B. Jones and Stefan Saroiu. Predictability requirements of a soft modem. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 37–49, New York, NY, USA, 2001. ACM Press.

[8] William Wong. Soft peripherals + hard cores = reconfigurable socs. In *Electronic Design*, July 2002.

[9] João Leonardo Fragoso, Eduardo Costa, Juergen Rochol, Sergio Bampi, and Ricardo Reis. Specification and design of an ethernet interface soft IP. January 01 2000.

[10] Srikanteswara S.; Reed J.H.; Athanas P.M.;. Implementation of a reconfigurable soft radio using the layered radio architecture. *Signals, Systems and Computers*, 1:360 – 364, Oct. 2000.

[11] Intel. Intel 8051 microcontroller. `www.intel.com`.

[12] ARC. Arc. `www.arc.com`.

[13] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[14] Fabrice Bellard. A generic linux soft modem. 1999.