Proyecto Fin de Máster

Trade-off between energy savings and execution time applying DVS to a microprocessor

Miroslav Vasić

Máster en Electrónica Industrial

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Departamento de Automática, Ingeniería Electrónica e Informática Industrial

Noviembre, 2007

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Departamento de Automática, Ingeniería Electrónica e Informática Industrial

Máster en Electrónica Industrial

Trade-off between energy savings and execution time applying DVS to a microprocessor

Autor: Miroslav Vasić

Director: Óscar García Suárez

Noviembre, 2007

Proyecto Fin de Máster

Index

Introducción

Una de las mayores preocupaciones para los diseñadores de sistemas electrónicos es el consumo de dispositivos. El bajo consumo reduce el precio de encapsulamiento y radiadores, y aumenta la autonomía y fiabilidad. Para los sistemas que utilizan baterías, el bajo consumo significa una vida más larga de la misma, menor precio del sistema y gran autonomía para los dispositivos móviles. La movilidad más alta es algo que se exige más y más cada día por los usuarios que quieren trabajar el mayor tiempo posible sin conexión con la red eléctrica. Una solución podría ser una batería más grande, pero esto significa más peso y menor movilidad. Entonces, la única solución será reducir el consumo del dispositivo.

Una de las técnicas utilizadas últimamente que disminuyen el consumo de microprocesadores es Escalado Dinámico de Tensión (Dynamic Voltage Scaling o DVS). Este método está basado en el hecho de que la parte mayor del consumo de microprocesadores depende de la tensión de alimentación como una función cuadrática:

$$P \propto CV_{DD}^2 f \tag{1a}$$

donde C es capacidad equivalente, V_{DD} es la tensión del núcleo y f es la frecuencia del reloj del sistema. Esta dependencia está basada en el modelo de una puerta lógica, donde la entrada de la puerta está presentada como un condensador. Por conmutación de la señal de la entrada, este condensador se carga y descarga y, por eso, la energía perdida es proporcional a la energía el condensador cuando está cargado.

Por tiempos de retraso de circuitos digitales, hay correlación entre la tensión mínima de alimentación y la frecuencia máxima del reloj de sistema y se puede representar como:

$$t_d = k\frac{V_{DD}}{(V_{DD} - V_{th})^\alpha} \propto \frac{1}{f_{MAX}} \tag{2a}$$

Donde k, V_{th} y a son constantes que dependen de la tecnología de circuitos CMOS. Por eso, para cada valor de la tensión de la fuente de alimentación existe una

frecuencia mínima del reloj de sistema, que garantice operaciones correctas del sistema digital.

Escalado Dinámico de Tensión (DVS) es la técnica que ofrece un ajuste de la tensión y de la frecuencia dependiendo de requisitos de tareas activas durante el tiempo de ejecución. Este ajuste se puede implementar de manera que se aproveche de los tiempos muertos de CPU, Figura 1a, o se puede reducir la velocidad de las tareas activas, Figura 2a.

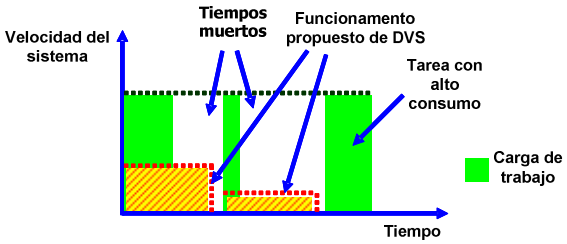

Figura 1a – DVS utilizando los tiempos muertos de CPU

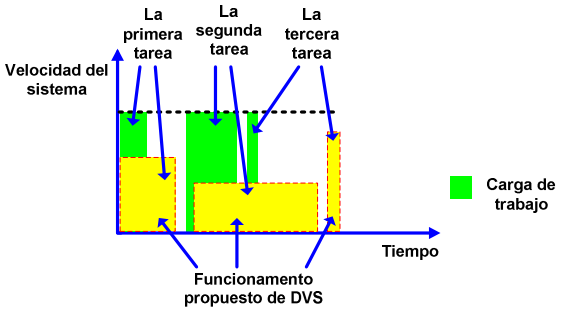

Figura 2a – DVS bajando la velocidad de las tareas activas

En el primer caso, tiempo muerto significa que la tarea activa se puede realizar con frecuencia más baja y por tanto la tensión de la alimentación se puede bajar también, Figura 1a. De esta manera, cada tarea estará realizada con la pareja de tensión y frecuencia apropiada para bajar el consumo sin reducción de prestaciones, utilizando todo el tiempo disponible.

La secunda solución está basada en el intercambio entre el tiempo de ejecución y ahorro de energía. Utilizando frecuencias bajas del reloj de CPU, y por tanto tensiones bajas de la fuente de alimentación, el tiempo de ejecución de tareas subirá, pero la energía de consumo bajará.

En los dos casos, los cambios de tensiones y frecuencias deberían ser rápidos para que no sean vistos por usuarios de sistema. El tiempo de transición necesario debería ser insignificante comparando con el tiempo de ejecución de las tareas activas. Por estas demandas, es necesario que exista comunicación entre el microprocesador y la fuente de alimentación con idea de que el microprocesador debería ser consciente cuando la transición esté acabada. La fuente de alimentación debería ser suficientemente rápida para apoyar estos cambios.

La idea de DVS ya está demostrada en algunos productos comerciales como los microprocesadores Crusoe y Efficieon de Transemta [1]. Estos microprocesadores están hechos para aplicaciones de bajo consumo y utilizan pasos de 33MHz para el reloj de sistema y escalones de 25 mV para la tensión del nucleó de CPU. La aplicación principal donde se pueden utilizar estos microprocesadores es en la descodificación de una imagen de video, aplicación en la cual ya se saben los patrones de actividad.

INTEL desarrolló su idea de DVS a través de "SpeedStep" y "Enhanced SpeedStep". La primera implementación, "SpeedStep", funcionó con sólo dos tensiones y frecuencias. La tensión y frecuencia más alta (1.7V y 1GHz) se utilizaban en el caso de un ordenador del usuario conectado con la red eléctrica, y la tensión y frecuencia más baja (1.1V y 500MHz) si el ordenador se alimenta desde su batería. Utilizando esta técnica es posible reducir el consumo de 34W hasta 8.1W [2]. La segunda solución utiliza varias tensiones y niveles de frecuencia para conservar la vida de batería mientras se mantiene el nivel alto de prestaciones. La tensión máxima para CPU es 1.4V y se utiliza con el reloj de 1.2GHz. Utilizando esta técnica, se alcanza un consumo medio de 22W. Para los valores mínimos de la tensión y de la frecuencia (1.1V y 700MHz) se necesita no más de 7W [3]

Otra solución para la técnica de DVS viene de AMD. Su implementación se llama "PowerNow!". El rango de las tensiones para el núcleo es de 1.4V hasta 1.8V, mientras la frecuencia se puede cambiar de 200MHz hasta 500MHz. Utilizando esta solución es posible alcanzar hasta un 75% de ahorros de energía [4].

Con respecto a otras implementaciones de la idea de DVS, en [5-9] se puede ver que los ahorros de energía varían entre 10% y más que 70%, dependiendo sobre todo de la cantidad de trabajo que sea necesaria.

Aparentemente, esta técnica ofrece muy buenos resultados, pero todavía, hay problemas que resolver (uno de ellos puede ser el conjunto finito de frecuencias de CPU). Hay cuatro partes importantes para implementar DVS en un sistema con microprocesador:

- Software (SW), incluyendo el sistema operativo y aplicaciones
- Fuente de alimentación (PS) que pueda generar las tensiones en un rango apropiado,

- Hardware (HW) que pueda funcionar en rango amplio de tensiones y
- DVS scheduler que elija la pareja de tensión y frecuencia apropiada, Figure 3a.

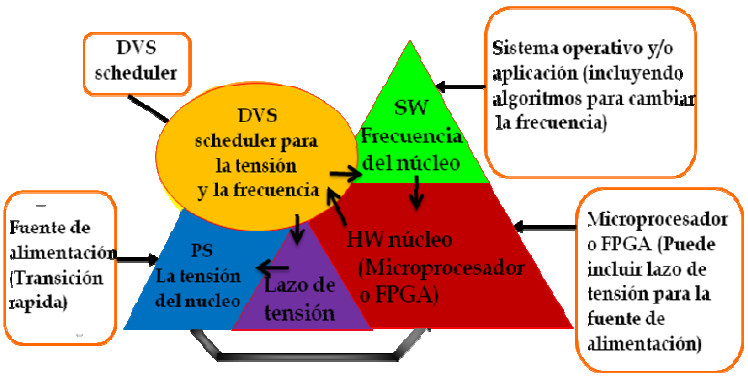

Figura 3a – Componentes importunes de un sistema de EDT

El objetivo del proyecto

El proyecto está motivado por las implementaciones de EDT mencionadas. El objetivo del proyecto es la implementación de un algoritmo de EDT para un microprocesador disponible comercialmente y la validación de la idea midiendo el consumo de energía.

El algoritmo propuesto se basa en un trade-off entre el tiempo de ejecución de la aplicación y la energía de consumo del microprocesador. Este trabajo propone un control del tiempo de ejecución usando los datos estadísticos obtenidos durante la actividad de la aplicación probada. Indirectamente, controlando el tiempo de ejecución también se controla la energía consumida. Un tiempo de ejecución más largo proporciona un menor consumo de la CPU.

Para implementar el sistema era necesario hacer el modelo del sistema que representa relaciones entre el tiempo de ejecución de la aplicación activa y la frecuencia del reloj de la CPU. Cuando el modelo estuvo hecho, el algoritmo de DVS se desarrolló e implementó en la plataforma.

Todas las pruebas se realizaron como las aplicaciones del "mundo real" para ver qué rango de ahorro de energía se podía obtener utilizando la propuesta técnica.

Finalmente, se discute sobre la influencia de algunas variables del sistema como el slew rate de la fuente de alimentación, el período de los cambios de la tensión y frecuencia, etc., en las prestaciones del sistema y la energía consumida por la CPU.

III.2.1 Cycle conserving algorithm

Before the end of a running process, we cannot say anything about the time it needs, so it is very natural to make very conservative assumption, that it will use his worst-case execution time. Every time when the process finishes earlier and gives us a little more time then it is planned, we can change operating frequency, because there is no need to work faster then it is necessary.

For EDF scheduling scheme it is very simple to implement this idea. First, it is necessary to define utility factor of a process. For process *i* utility factor is $U_i = \frac{C_i}{T_i}$ (C_i and T_i have been defined before). Now, it can be performed next pseudo code.

```
select frequency():
    use lowest freq.    f_i ∈ {f_1, ..., f_m | f_1 < ... < f_m}
    such that           U_1 + ... + U_n ≤ f_i/f_m
upon task release(Ti):
    set Ui to Ci/Pi
    select frequency();
upon task completion(Ti ):
    set Ui to cci/Pi    /* cci  is the actual cycles used this invocation */
    select frequency();
```

cc_i is actual execution time for process *i*, and the $\{f_1, f_2, \ldots, f_m\}$ are possible CPU frequencies. Article [29] contains detailed explanation of this algorithm and how it is implemented for a RM scheduled system. This technique can prevail up to 48% of energy saving in one EDF scheduled system [29].

III.2.2 Look Ahead algorithm (only for EDF schemes)

Previous algorithm assumes the worst possible situation, and if something is finished earlier then reduces the speed. With Look Ahead has been tried to make a little look into future and to postpone some processes as much as possible, in order to work with a minimum frequency [29]. Main idea is that the process with the highest priority should work until it ends, and that process with the lowest priority should wait as much as possible. Whenever we make assumption about the utility factor we presume worst case, i.e. that all processes work with the worst case time, and that we need to leave them space to finish their tasks, so the decision is pessimistic.

The pseudo code is the following one:

select_ frequency(x):
use lowest freq. $f_i \in \{f_1, \ldots, f_m | f_1 < \cdots < f_m\}$
such that $x \leq f_i / f_m$
upon task release(T_i):
set $c\_left_i = C_i$
defer();
during task_execution(T_i):
decrement c_left_i;
defer();
set $U = C_1 / P_1 + ... + C_n / P_n$;
set s=0;
for i=1 to n, $T_i \in \{T_1, ..., T_n | D_1 \geq ... \geq D_n\}$
set $U = U - C_i / P_i$;
set $x = max(0, c\_left_i - (1-U)(D_i - D_n))$;
set $U = U + (c\_left_i - x)/(D_i - D_n)$;
set s=s+x;
select_frequency ($s/(D_n$-current_time));

Where x is a number of the cycles which needs to be executed before the earliest deadline, after the process with the highest priority

This algorithm has a problem similar to the one in Cycle Conserving algorithm. When the process ends, it is necessary to change deadline, and to use new value for deadline of this process.

It is easy to see that the algorithms guarantee schedulability of the running processes. This algorithm is calculating utility factor for every moment of execution, and the idea is to reschedule this factor in time, so all task can be scheduled. Of course, this is possible only if they could be scheduled before applying DVS.

As a consequence of postponing the task with minimum priority level we can be in situation to work a long period of time with a maximum speed in order to end before deadline, and in those situations there will not be great power savings. Simulator results, which are presented in [29], ensure that this algorithm gains more then 50% of energy savings.

III.2.3 Algorithm based on properties of processed data (MPEG)

In some applications if we know the properties of data we process, like size of data block and approximate complexity of calculation, we can make decision about the voltage and frequency based on analyses of that information. This kind of algorithms are good for applications like MPEG and MP3 coding and decoding. The main idea will be explained on the MPEG decoding, [30].

A standard MPEG stream is composed of three types of compressed frames: I, P and B. "I" frames has information about one picture, while "P" and "B" frames are used for motion prediction and interpolation techniques between two successive "I" frames. Because of this it is clear that "I" frames are the biggest, and that "P" and "B" frames need much more calculations then "I" frame. Frames could be passed with different frequencies, i.e. frames per second. Standard values for this variable are: 23.976, 24, 25, 29.27, 30, 50, 59.94 and 60 frames per second. If the system is too fast for the selected frame speed, the decoder sets working mode to DELAY-PHASE, in order to make delay between successive frames. In the other hand, if the system is slow the decoder needs to skip some frames. Depending on the system slowness there are three different phases: B-PHASE, P-PHASE and I-PHASE. In the B-PHASE decoder discards some of the "B" frames, if this is not sufficient decoder skips some "P" frames, and that is P-PHASE. If even this is not good enough, then decoder starts to skip "I" frames, I-PHASE. In [30] there are two methods that are proposed for MPEG systems.

III.2.3.1 Delay and Drop Rate Minimizing Algorithm

Delay and Drop rate represent the main characteristics of the MPEG decoder. The ideal situation is to have Delay rate equal to zero and Drop rate too, this would mean that system is working with optimal speed. This algorithm tries to minimize both of these values. The pseudo code is the following.

```
static void scale-voltage-delay( )
{
    /*voltage is set to maximum voltage*/
    if((phase==I-PHASE) | | (phase ==P-PHASE))
            set-max-voltage ( ) ;
    else if (phase == B-PHASE)
            {       /* scale up voltage */
                    current_voltage = current_voltage+ drop_rate*increase_factor;
                    set-voltage(current-voltage);
            }
    else if(phase == DELAY-PHASE)
            {
                    if(delay<lOO)  /*keep current voltage*/
                                            return;
                    else { /* scale down voltage */
                            current_voltage =current_voltage - delay*decrease_factor;
                            set_voltage (current_voltage) ;
                    }
            }
}
```

III.2.3.2 Algorithm for predicting MPEG Decoding time

Previous algorithm is based on analyse of history, and sometimes doesn't work well, because MPEG decoding is RT application, and workload vary in time. These variations are present because of the different frame types and different scenes (static scenes and scenes with a movement). Before starting explanation of algorithm, it is necessary to define some variables:

- Weight_factor is used for controlling the relative weight of the most recent and past history of decoding time
- Balance_factor measures gap between the predicted workload and measured workload. If the Balance_factor is zero predictions is correct, if it is positive, then prediction causes delay by overestimating workload. If the value is negative then workload is underestimated
- DTPB is decoding time per Byte

Following is the pseudo code.

```
static void scale_voltage_predict( )
{
    update_statistics( );
    get_GOP_size( );
    decode_time= est_decode_time( );
    volt=set_volt_predict( decode_time );
    set_voltage(volt);
}


void update_statistics( )
{
    /* weighted average */
    avg_I_DTPB=avg_I_DTPB*(1-weight_factor)+ I_DTPB *weight_factor;
    avg_P_DTPB=avg_P_DTPB*(1-weight_factor)+P_DTPB*weight_factor;
    avg_B_DTPB=avg_B_DTPB*(1-weight_factor)+ B_DTPB * weight_factor;
}

void get_GOP_size( )
{
    I_size = get_GOP_I_size( );
    P_size = get_GOP_P_size( );
    B_size = get_GOP_B_size( );
}

double est_decode_time( )
{
    I_decode_time = I_size * avg_I_DTPB;
    P_decode_time = P_size * avg_P_DTPB;
```

```
        B_decode_time = B_size * avg_B_DTPB;
        return I_decode_time + P_decode_time + B_decode_time;
}

void set_volt_predict( d_time )
{
        if((phase==I_PHASE)||(phase==P_PHASE))
                balance_factor = I_P_balance;
        else if(phase == B_PHASE)
                balance_factor = drop_rate*B_balance;
        else if(phase == DELAY_PHASE){
                balance_factor = delay*DELAY_balance;

        /*scaled well, thus no volt. change occurs*/
        if(delay<100)
        {
        /*standard voltage and time are set to current values to adjust to current workload*/
                std_voltage = voltage;
                std_dtime = d_time;
        }

        return std_voltage*d_time/std_dtime+ balance_factor;
}
```

In this algorithm problem is that weight_factor constant value for each MPEG application and needs to be set before the DVS is started.

The both MPEG algorithm shown good results with power savings from 15% to 80%, with low number of lost frames, and savings mainly depending on the complicity of the images.

Implementation of one DVS system

In order to validate possible power savings achieved by DVS technique, a platform based on Intel's XScale PXA255 was used for the implementation of a DVS algorithm. The used platform is Arcom's Viper board [31]. The Viper board is actually an ultra low power, arm compatible, PXA255 based computer for power sensitive embedded communications and multimedia applications. The board has several serial and USB ports and one Ethernet port, as well, which can be used for communication between a user and the system, Figure 11.

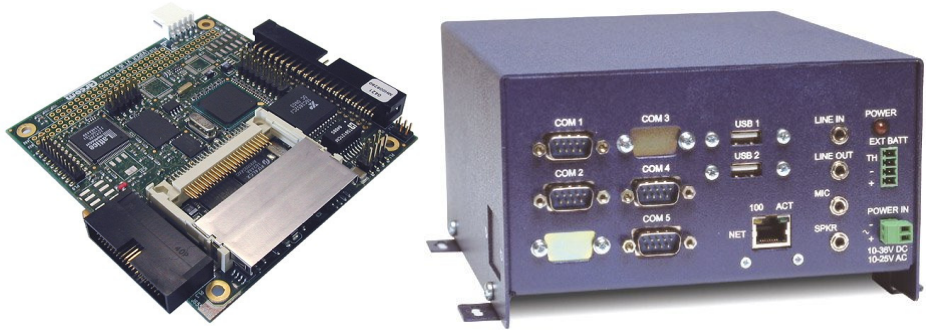

Figure 11 Viper board with Intel PXA255 microprocessor

The operating system, which is originally installed, can be Windows CE 5.0 or Linux 2.6.17. This board was chosen for two reasons. The first is, obviously, the microprocessor that is meant to be used in a DVS application. The second is the system itself, because it represent a relatively good model of "real world" system (the board is not made for specialized application, it is generally purpose computer), hence, all the conclusions about the things which are necessary for the implementation of DVS and about the influence on the DVS performance on this board can be generalized.

In the following paragraphs is detailed description of the used board, its power supply system and operating system.

IV. Viper XScale PXA255 board

The Viper board consists of the microprocessor, PXA255, 32KB of data cache, and 32KB of instruction cache. There are 64MB of SDRAM and 32MB of additional Flash memory. The system can be booted with its File System on the Flash, or as Network File System, when all the files and data on the remote PC. A system user can accede to the Flash memory through the Ethernet or Serial, RS-232, connection. The board does not have its screen, so the "graphical interface" with a user is done through the user's PC and Hyper Terminal, if the user uses Windows as his operating system, or using Minicom or Secure Shell network protocol, for Linux based computers. In Figure 12 the block diagram of the Viper board [32] is shown.

The power supply of the board is specialized supply form Maxim, MAX1702 [33]. As it can be seen in Figure 12, this power supply has three outputs. The outputs need to supply CPU, memory and I/O pins of the board. For a DVS implementation the most important information is the one about the voltage supply for the core. The board is originally designed to supply the core from 1.06V to 1.29V, depending on the applied frequency of the system clock.

To adjust the core's voltage, CPU needs to write a certain value to the Micropower DAC from figure 12. The used DAC is LTC1659 DAC and this DAC has a serial interface with PXA255. The purpose of this DAC is to set the reference for the output voltage of the MAX1702 core supply. For each change of the reference it is necessary to send 12 bits. When changing the core voltage it is important to ensure that the internal CPU clock is set to the correct voltage range. The CPU core supply must be set to a defined range for a particular clock. In the Table 4 the values of DAC that are recommended to be used [32] are shown.

DAC Data Hex Value	CPU Core Voltage	Comment
0x000	1.65V	Not recommended to set the VCORE above 1.3V as the power consumption will increase for no performance benefit
0x325	1.29V	Typical VCORE for peak voltage range at 400 MHz operation. Maximum VCORE for medium voltage range at 200MHz
0xDE5	1.1V	Typical VCORE for high voltage range at 300MHz
0xFFF	1.06V	Typical VCORE for low voltage and medium voltage range, suitable for 100MHz to 200MHz operation

Table 4 – The values of the power supply's DAC and corresponding supply's voltage

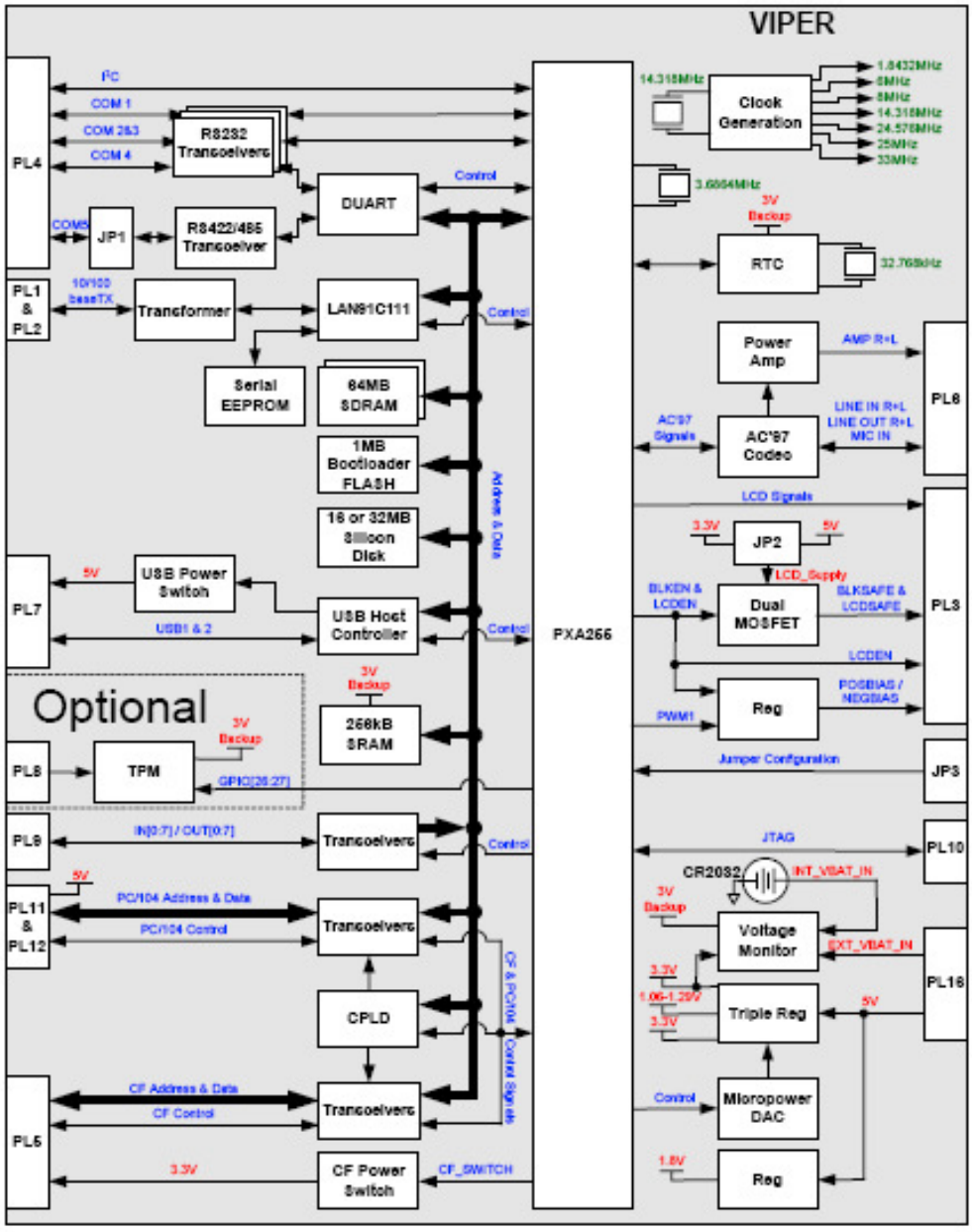

Figure 12 – Block diagram of the Viper board

As it can be seen, by default, the core's voltage is same in the case of 100MHz and 200MHz system clock.

It is important to know that any change of CPU's clock frequency means the change of internal (f_{INT}) and external (f_{EXT}) bus frequency, as well. The internal bus connects the core and other functional blocks inside the CPU such as I/D-cache unit and the memory controller whereas the external bus in the target system is connected to SDRAM (64MB). This correlation yields to conclusion that the frequency of those two buses has influence on the application performance and execution time. The Table 5 presents the set of possible frequencies, with corresponding default voltages, which can be used on the Viper board.

	f_{CPU} (MHz) core	f_{INT} (MHz) internal bus	f_{EXT} (MHz) external bus	Core voltage (V)
f1	100	50	100	1.06
f2	200	100	100	1.06
f3	300	100	100	1.1
f4	400	200	100	1.3

Table 5 – The frequencies and the voltages used on the Viper board

Figure 13 shows block diagram of the connection between the CPU, the DAC and the power supply.

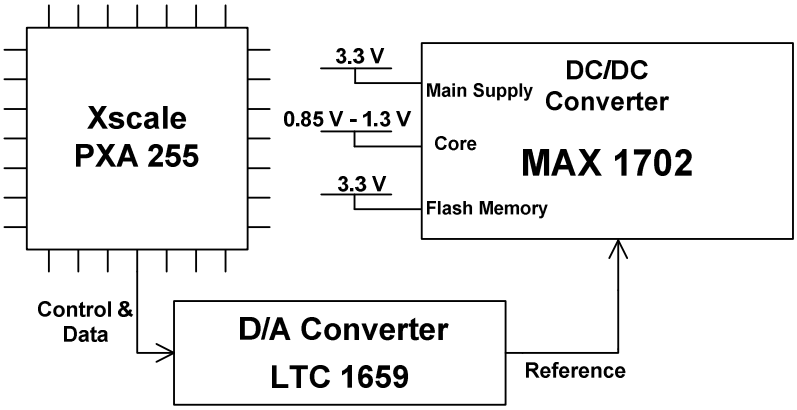

Figure 13 – Block diagram of the connection between the CPU, the DAC and the power supply

Figure 14 presents detailed schematic of the core's voltage control.

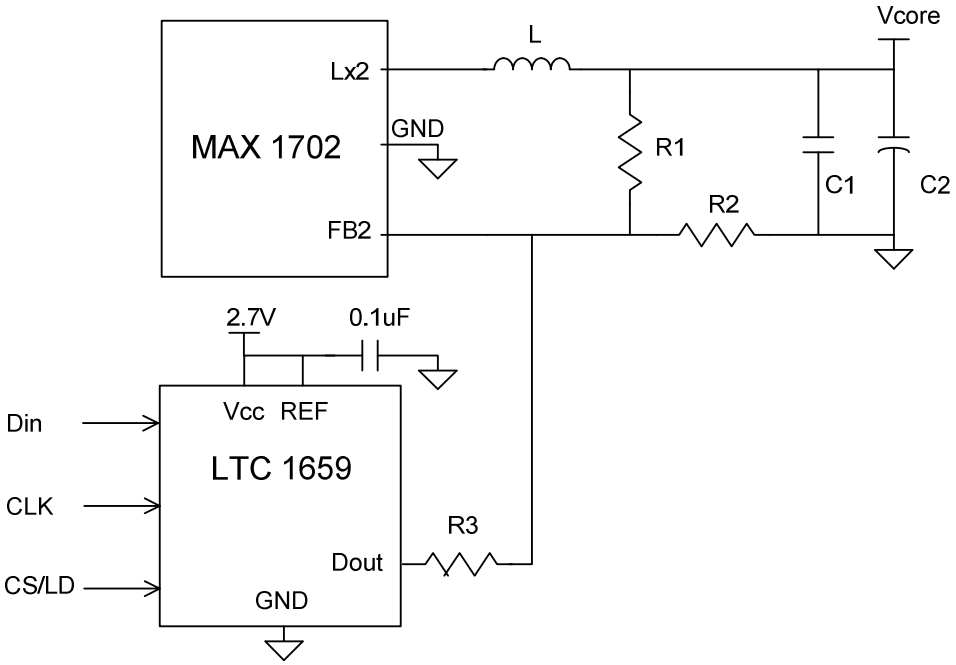

Figure 14 – Detailed schematic of core's voltage control

The elements L, C_1 and C_2 are output filter of the MAX1702, while the desired voltage is set by resistor divider consisted of R_1, R_2 and R_3. In steady state the voltage between the pin FB_2 and the ground is 0.7V, and should remain between 686mV and 714mV. If there were not be the resistor R_3 the output voltage would be:

$$V_{OUT} = 0.7V * (1 + \frac{R_1}{R_2})$$

Because of the additional resistor, the output voltage is given by:

$$V_{OUT} = 0.7V * (1 + \frac{R_1}{R_2} + \frac{R_1}{R_3}) - \frac{R_1}{R_3} V_{DAC}$$

Where V_{DAC} is the output voltage of the D/A converter.

In the case when the voltage of FB_2 is beyond its boundaries, the regulator of the corresponding output voltage thinks that the regulation has been lost and that output will be shut off and, therefore, CPU will stop its work. This can happen during the voltage transitions of the output voltage of the DAC. Due to this problem, voltage transitions must be done step-by-step, not instantly. It is recommended to use steps of 0x100 at DAC, which is, approximately, 20mV of output voltage. In

Figure 15 one transition from 1.06V to 1.29V is shown. There can be noticed 13 steps that are needed for this change.

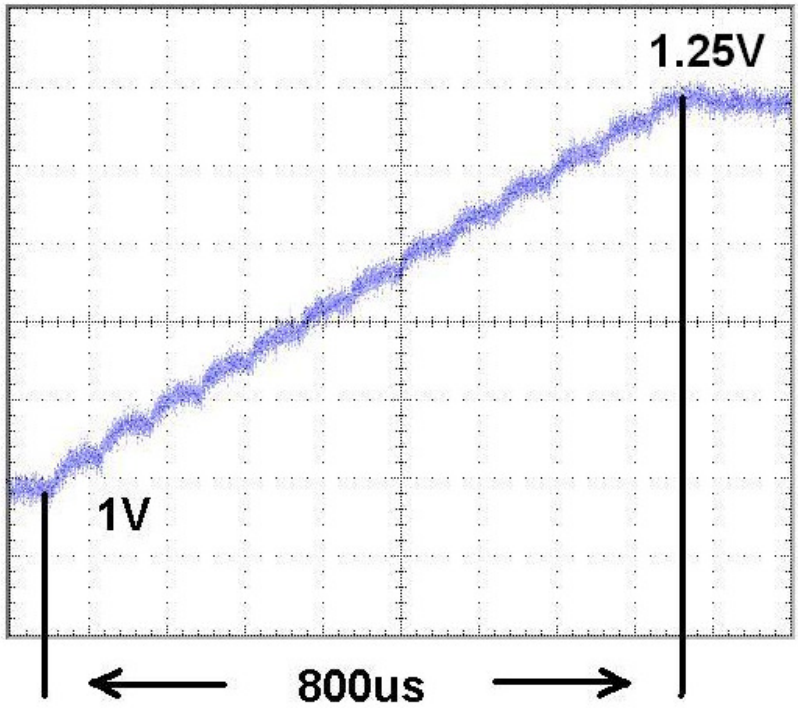

Figure 15 – Transition of core's voltage from 1V to 1.3V

The operating system that is used, by default, is Linux 2.6.17. By activating already made software modules, the user can manually change frequency of the clock and the voltage of the microprocessor from the command line. On the software level these changes are done on high level of abstraction. As it was aforementioned, in order to change frequency of the core's clock it is necessary to change data in Core Clock Control Register (CCCR). This register contains information about the frequency multipliers that are used in order to produce demanded frequency for the core's clock. Figure 16 presents Clocks Manager Block diagram with clock distribution for XScale PXA255.

Of course, the one who wants to change the core's frequency does not need to know anything about the CCCR register. The wanted frequency is the only information he needs. Hence, some software structures as driver, policy and governor for voltage and frequency changes are made. These structures simplify any frequency/voltage change from the software level. Figure 17 shows the hierarchy between these software structures.

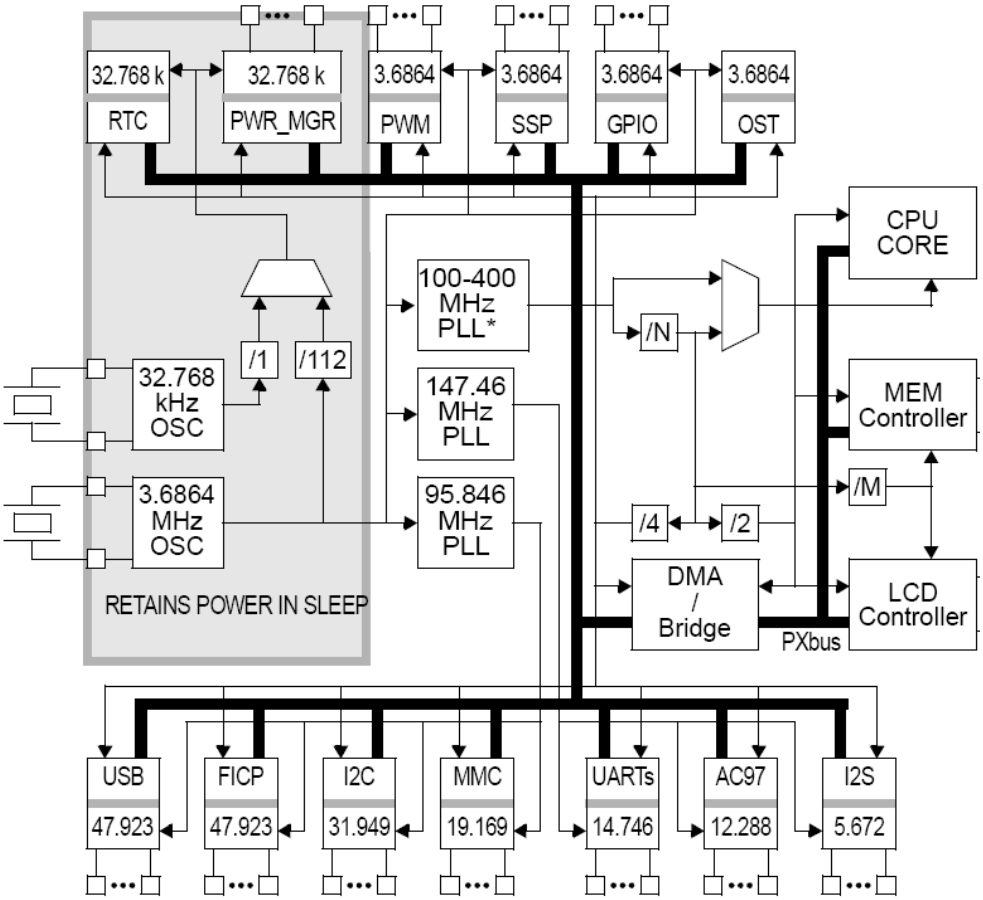

Figure 16 – Block diagram of Clock Manager for Intel's PXA255

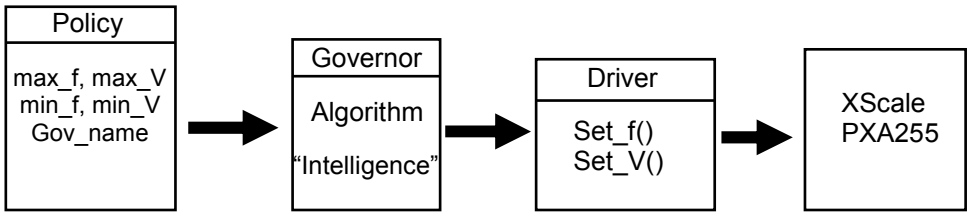

Figure 17 – The hierarchy between software structures for voltage and frequency changes

The driver is the lowest level of abstraction, and its task to communicate directly with CPU, doing all the "dirty" work. Other structures that can use this driver use only the functions to give an order for frequency and voltage change, and the driver

sets the corresponding values in CCCR. Thanks to the structure it is not necessary to know hexadecimal values that are needed to be written in the register, the driver does it on assembly level.

The policy is the highest level of interface with a system user. On this level user makes decision about the minimal and maximal frequency which can be used in the implemented DVS system and it is necessary to specify which DVS algorithm will be used. The DVS algorithm is implemented on the level of Governor, which is between the driver and the policy, and it is a link between these two levels. The Governor makes decision which frequency within the minimal and maximal should be used. One DVS system always has one policy, but can have several governors, which can be changed dynamically depending on users need.

If we look on the block diagram of the software structure hierarchy, we can see which parts of the system can improve the DVS performance. On the first place is the algorithm, the governor level. Depending on the algorithm, power savings can differ significantly. In order not to hold back with CPU's work, the transitions of voltage and frequency should be very fast, and if not, this can be seen in lower performance of the system and its energy consumption without any useful activity. Thus, the dynamics of the PLL, which is used to generate clock frequency and the core's power supply, are something to think about. Unfortunately, the PLL is realized inside the CPU, so it cannot be improved, but the power supply sure it can be. Hence, the DVS algorithm and power supply are main constraints in our system.

V. Implemented DVS Algorithm

The implemented DVS algorithm is based on the trade-off between the execution time and the power savings. By using lower frequencies of the CPU clock, and therefore lower supply voltages, application's execution time will increase, but the energy consumption will be lower. A system user should select the performance loss of execution time he is willing to sacrifice in order to decrease power consumption.

The proposed algorithm is based on the work in [6] and the main idea is decomposition of the CPU's work in order to control the execution time of the running application. The algorithm needs to estimate the time that the active task requires to execute instructions inside the CPU only and the time spent by the task in the communication between the CPU and the memory. The estimation is done using the model that illustrates the correlation between the execution time of the active task, its memory activity and CPU frequency, and it is presented in the next paragraph.

V.1 Workload decomposition

Execution time, which software program needs, can be represented in term of cycles per instruction (CPI) and CPUs' clock frequency as follows [34]:

$$T = \frac{\sum_{i=1}^{n} CPI_i}{f^{CPU}} \tag{3}$$

where n is the total number of executed instructions, CPI_i is the number of CPU clock cycles for the i^{th} instruction and the CPU's frequency f^{CPU}. Naturally, this is only in the case if all the instructions are executed only inside the CPU. The problem rises in the moment when it is necessary to fetch the data or the instruction that is not in cache memory. This situation yields to stall cycles of the CPU (cycles when the CPU is not active) and it is referred as stall cycles due to I/D cache miss or off-chip stall cycles. While the CPU is halted, the instruction or the data is taken from the memory, and the time needed for this operation is called off-chip time. Stall cycles can be produced, as well, by data/control dependency or branch misprediction on CPU. The stall cycles produced by these events are on-chip stall cycles.

Workload of a task is defined as the sum of the CPIs of all instructions of the software program, hence the workload of a program can be defined as [6]

$$W = N \cdot CPI^{avg} = N \cdot (CPI_0 + CPI^{avg}_{branch\_miss} + CPI^{avg}_{stall\_onchip} + CPI^{avg}_{stall\_offchip}) \tag{4}$$

where N is the number of instructions, CPI_0 is the ideal CPI which is 1 for a single-issue general-purpose microprocessor, $CPI^{avg}_{branch\_miss}$ stands for the number of CPU clock cycles due to branch misprediction overhead, and $CPI^{avg}_{stall\_onchip}$ and $CPI^{avg}_{stall\_offchip}$ are the numbers of CPU clock cycles due to on-chip stalls and off-stalls, respectively.

Let us define two types of workload: on-chip workload and off-chip workload.

On-chip workload, W^{ON}, is the number of CPU clock cycles required to perform the set of on-chip instructions, which are executed inside the CPU only. The execution time required to finish W^{ON}, T^{ON}, depends only on the CPU frequency, f^{CPU}, and it is calculated as

$$T^{ON} = \frac{W^{ON}}{f^{CPU}} \tag{5}$$

Off-chip workload, W^{OFF}, is the number of external clock cycles needed to perform the set of off-chip accesses. The execution time required to complete W^{OFF}, T^{OFF}, depends on the external and internal bus frequency, and it is calculated as [6]

$$T^{OFF} = \frac{(1-\alpha)\cdot W^{OFF}}{f^{INT}} + \frac{\alpha \cdot W^{OFF}}{f^{EXT}} \tag{6}$$

where a is a constant and $\alpha \in (0,1)$, f^{INT} and f^{EXT} are internal and external bus clock frequency, respectively.

T^{OFF} is calculated in this way because of the data fetching mechanism. For example, data cache miss requires data fetch from external memory and data transfer to the CPU core. For the first action the frequency of external bus has influence on T^{OFF} time, and for the second the clock of the internal bus. Therefore, in order to simplify the model of T^{OFF} time, the a is introduced as the ratio between the time which is spent due to data transfer on the internal bus and the time spent by the external bus cycles.

Using earlier definitions, it can be written:

$$W^{ON} = N \cdot CPI_{on}^{avg} = N \cdot (CPI_0 + CPI_{branch\_miss}^{avg} + CPI_{stall\_onchip}^{avg}) \tag{7}$$

$$W^{OFF} = N \cdot CPI_{stall\_offchip}^{avg} = M \cdot CPI_{off}^{avg} \tag{8}$$

where CPI_{on}^{avg} is the number of CPU clock cycles per on-chip instruction, M is the number of off-chip accesses, and CPI_{off}^{avg} is the number of external clock cycles per an of-chip accesses. From these two definitions, the execution time of running task, T, is presented as:

$$T = T^{ON} + T^{OFF} = \frac{N \cdot CPI_{on}^{avg}}{f^{CPU}} + \frac{\alpha \cdot M \cdot CPI_{off}^{avg}}{f^{INT}} + \frac{(1-\alpha)\cdot M \cdot CPI_{off}^{avg}}{f^{EXT}} \tag{9}$$

It is necessary to define next variables, too:

$$X = \frac{f^{CPU}}{f^{EXT}} \qquad Y = \frac{f^{CPU}}{f^{INT}} \qquad \beta = \frac{T^{OFF}}{T^{ON}} = \frac{W^{OFF}}{W^{ON}} \cdot (X\alpha + Y\alpha^{'}) \ , \ \alpha^{'} = 1 - \alpha \tag{10}$$

It can be seen that X represents the ratio of the frequency of the CPU and the frequency of the external data bus, Y stands for the ratio of the CPU's frequency and the frequency of the internal bus and β denominates the ratio of the time needed by off-chip accesses and the time spent by chip-on instructions.

Using last definitions, it can be written:

$$T^{OFF} = \frac{W^{OFF}}{f^{CPU}} \cdot (X\alpha + Y\alpha^{'}) \tag{11}$$

Due to the increased execution time it is necessary to define the application's performance loss as follows:

$$PF = \frac{T_{f^{CPU}}}{T_{f_{MAX}^{CPU}}} - 1 \tag{12}$$

$$PF \cdot T_{f_{MAX}^{CPU}} = T_{f^{CPU}} - T_{f_{MAX}^{CPU}} \tag{13}$$

where $T_{f_{MAX}^{CPU}}$ stand for the execution time at the CPU's maximum frequency, and $T_{f^{CPU}}$ is the execution time at the CPU frequency of f^{CPU}. Thus, PF shows how much the execution time of the tested application is longer than the time when the application is executed with maximal speed. For example, in the case of PF=0,2, execution time of the application with the frequency of CPU's clock of f^{CPU} is 20% longer than the time in the case when the maximal frequency is applied.

The key idea of the algorithm is to estimate $T^{ON}_{f_{MAX}^{CPU}}$ and $\frac{T^{OFF}}{T^{ON}}$ or β, as it is defined in equation 10. Using these two estimated values it is possible to estimate $T_{f_{MAX}^{CPU}}$. In [6] is proposed an algorithm where the estimation is done using statistic data from the microprocessor. Data such as number of data cache misses (DPI), number of stall cycles (SPI) and number of executed instructions per instruction (CPI). In this work is presented similar algorithm as in [6], but with a model of execution time that has in mind the relationship between the frequencies of the CPU clock, external and internal bus. Additionally, the solution in this work has in mind hardware limitations, i.e. finite number of possible frequencies that can be chosen. The finite number of CPU frequencies is recompensed by the feed back information about the

estimated execution time made after the CPU frequency is chosen. In the solutions from [5] and [6], the algorithm works in the open loop, without the feedback. Figure 18 shows the block diagram of the proposed algorithm.

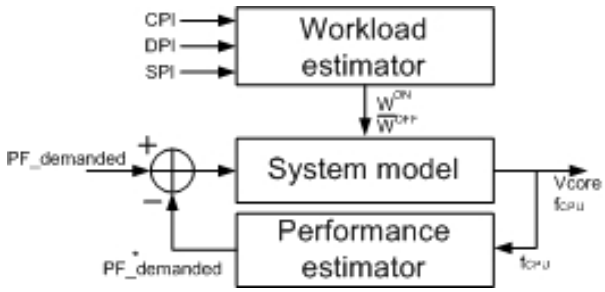

Figure 18 – The block diagram of the proposed algorithm

Very important step is to make a presumption that used statistics does not have a very high dynamic. In another words, if we observe the running task in one time period, its statistic in that period will be, more or less, similar to its statistics from the next period. As it can be seen, the algorithm is trying to foresee the activity of the tested application looking in its past and apparently every time when there is huge change of activity it will make an error in prediction. This error is one of the drawbacks of this algorithm. So, let us presume that β and $T^{ON}_{f^{CPU}_{MAX}}$ have been estimated, and therefore $T_{f^{CPU}_{MAX}}$ (the algorithm how it is done will be explained later). If so, for a desired performance loss, *PF*, and estimated $T_{f^{CPU}_{MAX}}$ (it will be denoted as $T'_{f^{CPU}_{MAX}}$) it can be written:

$$PF \cdot T'_{f^{CPU}_{MAX}} = T_{f^{CPU}} - T_{f^{CPU}_{MAX}} \tag{14}$$

Doing this, the "past" of the active application is separated on the left side of the equation and the "future" is on the right side.

Now it is necessary to calculate f^{CPU} for the given *PF* and estimated β.

For a given W^{OFF} and using the data from table 5 and equations (6), (10) and (11), as well, it can be written:

$$T^{OFF}_{f^{CPU}_{MAX}} = T^{OFF} \cdot \frac{f^{CPU}}{f^{CPU}_{MAX}} \cdot \frac{4 \cdot \alpha + 2 \cdot \alpha'}{X \cdot \alpha + Y \cdot \alpha'} = \beta \cdot \frac{W^{ON}}{f^{CPU}} \cdot \frac{f^{CPU}}{f^{CPU}_{MAX}} \cdot \frac{4 \cdot \alpha + 2 \cdot \alpha'}{X \cdot \alpha + Y \cdot \alpha'} \tag{15}$$

where T^{OFF} is the off-chip time of the application when f^{CPU} is used as a core's clock frequency and X and Y are corresponding ratios of internal and external bus frequencies and the core's frequency, as earlier defined.

Developing the left side of equation (14) and using equations (5), (6) and (15), it is obtained:

$$PF \cdot T'_{f^{CPU}_{MAX}} = PF \cdot \left(\frac{W^{ON}}{f^{CPU}_{MAX}} + \beta \cdot \frac{W^{ON}}{f^{CPU}_{MAX}} \cdot \frac{4\alpha + 2\alpha'}{X'\alpha + Y'\alpha'}\right) = PF \cdot \frac{W^{ON}}{f^{CPU}_{MAX}} \cdot \left(1 + \beta \cdot \frac{4\alpha + 2\alpha'}{X'\alpha + Y'\alpha'}\right) \quad (16)$$

For example, let us assume that the estimated values of β and $T^{ON}_{f^{CPU}_{MAX}}$ are 4.56 and 1.8ms respectively and that in the time quantum of 10ms (when the statistics data were collected and the estimation was done) the CPU frequency was 300MHz and that a of the tested application is 0.9. Having in mind these information and the data from the table 5, $T'_{f^{CPU}_{MAX}}$ is estimated, using equations (9) and (16):

$$T'_{f^{CPU}_{MAX}} = 1.8ms \cdot \frac{300MHz}{400MHz} \cdot \left(1 + 4.56 \cdot \frac{4 \cdot 0.9 + 2 \cdot 0.1}{3 \cdot 0.9 + 3 \cdot 0.1}\right) = 9.1476ms$$

This means that the workload that was processed in 10ms with 300MHz system clock would have been processed in 9.1476ms if the frequency of 400MHz had been used. Multiplying this value with demanded time performance loss we obtain the amount of time for which we need to slow down the application in the next time quantum. If the *PF* is 40% that means that we need to choose frequency which will process the same workload 3,659ms slower comparing with the time needed when the maximal frequency of the CPU clock is applied.

Next step is to calculate the needed frequency.

Developing the right side of (14) using equations (5), (6) and the data from table 5:

$$T_{f^{CPU}} - T_{f^{CPU}_{MAX}} = \frac{W^{ON}}{f^{CPU}} - \frac{W^{ON}}{f^{CPU}_{MAX}} + \frac{W^{OFF}}{f^{CPU}} \cdot (X\alpha + Y\alpha') - \frac{W^{OFF}}{f^{CPU}_{MAX}} \cdot (4\alpha + 2\alpha') \quad (17)$$

$$T_{f^{CPU}} - T_{f^{CPU}_{MAX}} = \frac{W^{ON}}{f^{CPU}} - \frac{W^{ON}}{f^{CPU}_{MAX}} + \beta \cdot \frac{W^{ON}}{f^{CPU}} - \beta \cdot \frac{W^{ON}}{f^{CPU}_{MAX}} \frac{4\alpha + 2\alpha'}{X\alpha + Y\alpha'} \quad (18)$$

Joining the developed sides of equation (14):

$$PF \cdot \frac{W^{ON}}{f^{CPU}_{MAX}} \cdot \left(1 + \beta \cdot \frac{4\alpha + 2\alpha'}{X'\alpha + Y'\alpha'}\right) = \frac{W^{ON}}{f^{CPU}} - \frac{W^{ON}}{f^{CPU}_{MAX}} + \beta \cdot \frac{W^{ON}}{f^{CPU}} - \beta \cdot \frac{W^{ON}}{f^{CPU}_{MAX}} \frac{4\alpha + 2\alpha'}{X\alpha + Y\alpha'} \quad (19)$$

$$PF \cdot (1+\beta \cdot \frac{4\alpha + 2\alpha^{'}}{X^{'}\alpha + Y^{'}\alpha^{'}}) = \frac{f_{MAX}^{CPU}}{f^{CPU}} - 1 + \beta \cdot \frac{f_{MAX}^{CPU}}{f^{CPU}} - \beta \cdot \frac{4\alpha + 2\alpha^{'}}{X\alpha + Y\alpha^{'}} \quad (20)$$

$$PF + 1 + \beta \cdot (4\alpha + 2\alpha^{'})[\frac{PF}{X^{'}\alpha + Y^{'}\alpha^{'}} + \frac{1}{X\alpha + Y\alpha^{'}}] = \frac{f_{MAX}^{CPU}}{f^{CPU}}(1+\beta) \quad (21)$$

$$f^{CPU} = \frac{f_{MAX}^{CPU}(1+\beta)}{\mathrm{PF} + 1 + \beta(4\alpha + 2\alpha^{'})(\frac{PF}{X^{'}\alpha + Y^{'}\alpha^{'}} + \frac{1}{X\alpha + Y\alpha^{'}})} \quad (22)$$

The final equation, that expresses f^{CPU} needed for the given PF and estimated β, clearly shows the influence of internal and external bus on the frequency that should be elected. In [6] the influence of these two buses is neglected and f^{CPU} is calculated as follows

$$f^{CPU} = \frac{f_{MAX}^{CPU}}{1 + \mathrm{PF}(1 + \beta \frac{f_{MAX}^{CPU}}{f^{CPU}})} \quad (23)$$

Thus, estimating β, and knowing the performance loss we are willing to sacrifice we are able to calculate the frequency of the CPU's clock by the formula (22).

Proposed frequency (MHz)	X	Y	The error made by electing the proposed frequency (MHz)
100	1	2	23
200	2	2	37
300	3	3	107
400	4	2	192

Table 6 – The error made by electing the frequency of the CPU clock that is the closet to the theoretical value

When the frequency is calculated, the next step is to choose the frequency of the CPU clock that is the most approximate to the calculated one. The equation (22) is not given in explicit form, because on its left side is the frequency of the CPU, and on its right side X and Y that depend on that frequency. Hence, the frequency that should be elected is the one that has the minimal error defined as:

$$error = \left| f^{CPU} - \frac{f_{MAX}^{CPU}(1+\beta)}{\mathrm{PF}+1+\beta(4\alpha+2\alpha')(\frac{PF}{X'\alpha+Y'\alpha'}+\frac{1}{X\alpha+Y\alpha'})} \right| \quad (24)$$

For example, if the demanded time performance loss were 40%, a of the tested application were 0.9, the CPU frequency in the last quantum were 300MHz and estimated β were 1.15 we would choose 100MHz , Table 6.

V.2 Algorithm to estimate $T^{ON}_{f^{CPU}_{MAX}}$ and β

In order to estimate $T^{ON}_{f^{CPU}_{MAX}}$ and β it is, actually, necessary to estimate W^{ON} and W^{OFF} for an observed period of time. This estimation is done using the statistics about the activity of running task. XScale family of microprocessors is supplied with Performance Monitoring Unit (PMU) which is consists of two programmable counters. The PMU supports monitoring of 15 performance events including cache misses, number of stall cycles and number of executed instructions. The overhead for accessing the PMU is 1μs and for this implementation can be ignored. Only limitation is that due to counters only two events can be monitored. Except those two programmable counters, there is the third one, which always counts the number of clock cycles within a desired period of time (CCNT).

As it was said, earlier, there are three important events to be monitored:

- number of executed instructions (INSTR), needed to obtain the value of CPI
- number of stall cycles (STALL), the number of on-chip and of-chip CPU cycles
- number of data cache misses (DMISS)

If it was used XScale 80200 it would be enough to use only INSTR and DMISS [5]. The architecture of XScale PXA255 permits "miss-under-miss". When a data cache miss requests data in the same cache line as a previous data cache miss event, then an external memory access does not occur for the current data cache miss, but it is counted as it was done.

Due to the limitation of number observed events, the PMU will be read twice in observed quantum of time. In the begging of each quantum of time first will be read INST, STALL and CCNT. When the half of the quantum has passed, the value of DMISS and CCNT will be read. It is presumed that the CPU statistics from one half of quantum are the same in the whole quantum. In figure 19 DMISS in time for a "gzip" application is presented. The red trace is DPI taken each 10ms, and the blue one is DPI taken each 5ms. It can be seen that there are no significant difference

between those two traces, except that the peak values of the blue trace are little bit higher, due to lower averaging time. Hence, we can conclude that our presumption can be used for this application.

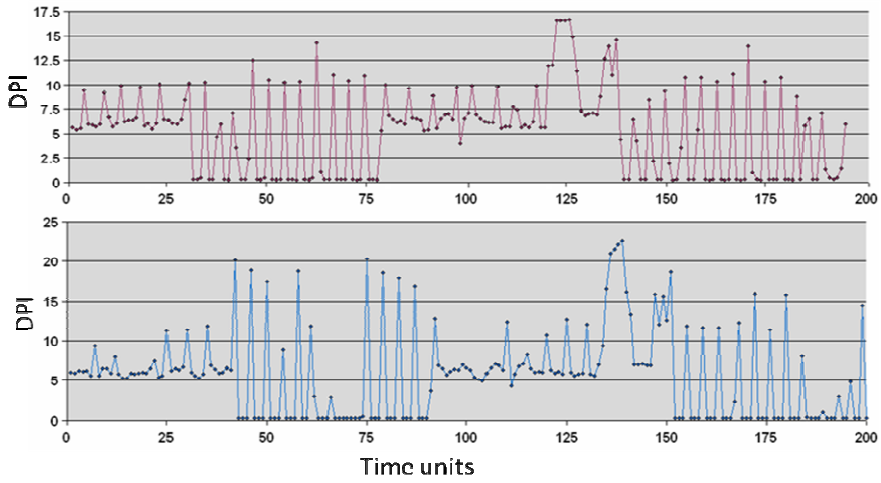

Figure 19 – Time diagram of DPI taken for gzip application with periods of 10ms (the red trace) and 5ms (the blue trace)

As it was before presented, for CPI^{avg} stands:

$$CPI^{avg} = CPI_0 + CPI^{avg}_{branch\_miss} + CPI^{avg}_{stall\_onchip} + CPI^{avg}_{stall\_offchip} \tag{24}$$

$$CPI^{avg} = CPI_0 + CPI^{avg}_{branch\_miss} + SPI^{avg} \cong 1 + SPI^{avg} \tag{25}$$

Using CCNT, STALL and INSTR it can be calculated following:

$$CPI^{avg} = \frac{CCNT}{INSTR} \qquad SPI^{avg} = \frac{STALL}{INSTR} \tag{26}$$

Figure 20 presents the dependency between CPI^{avg} and SPI^{avg} for "gzip" application. Each point represents one reading of PMU. The linearity and congruence with theory is obvious.

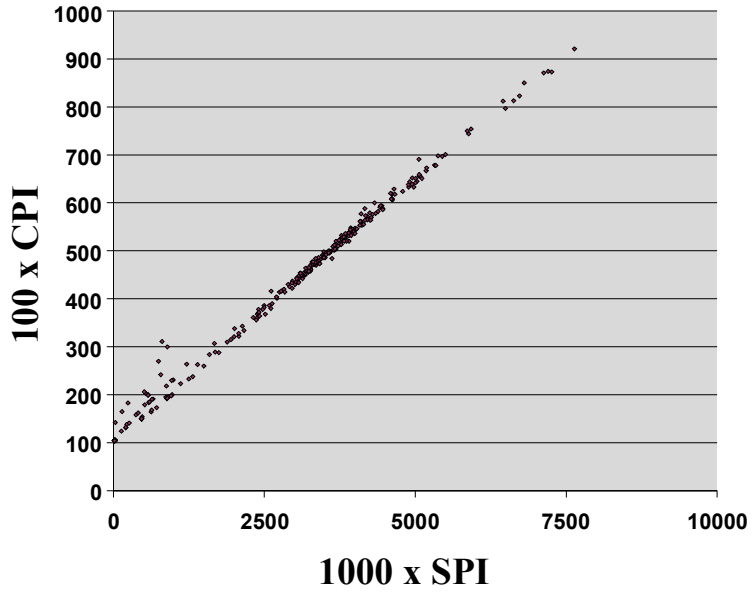

Figure 20 – Relationship between CPI and SPI for gzip application

The interception of the interpolated line and CPI axis is equal to average on-chip CPI without any stall cycles and will be denoted as $CPI_{on}^{\min}$. The slope value and the value of intersection between the obtained line and CPI axis are calculated using regression equation. For linear functions given by $y=b*x+c$, the coefficients b and c at instant (quantum) t, using last N measures of x and y, are calculated as:

$$b = \frac{N \cdot (\sum_{i=t}^{t-N+1} x_i \cdot y_i) - (\sum_{i=t}^{t-N+1} x_i) \cdot (\sum_{i=t}^{t-N+1} y_i)}{N \cdot (\sum_{i=t}^{t-N+1} x_i^2) - (\sum_{i=t}^{t-N+1} x_i)^2} \tag{27}$$

$$c = \frac{(\sum_{i=t}^{t-N+1} y_i)}{N} - b \cdot \frac{(\sum_{i=t}^{t-N+1} x_i)}{N} \tag{28}$$

where $t \geq N$, x_i and y_i stand for SPI^{avg} and CPI^{avg} at i^{th} quantum, respectively.

To obtain $CPI_{on}^{avg} = CPI_{on}^{\min} + SPI_{on}^{avg}$ it is required to extract SPI_{on}^{avg} from SPI^{avg}. In [6] was explained how this can be done using the information about the quantum's DMISS. The variables used for the extraction of CPI_{on}^{avg} are shown in Figure 21. $CPI_{on}^{\max}$ is the lowest value of CPI_{on}^{avg} obtained during the application's execution.

If there is not any data cache miss or they are low, the value of CPI_{on}^{avg} is close to $CPI_{on}^{\max}$ because all the stall cycles comes from the CPU. In contrary, when the value of data cache misses is high, CPI_{on}^{avg} is close to $CPI_{on}^{\min}$. Thus, CPI_{on}^{avg} always will be between $CPI_{on}^{\min}$ and $CPI_{on}^{\max}$, depending on the data cache miss rate and it is necessary to obtain CPI_{on}^{avg} using the information about DPI^{avg}. DPI^{avg} is calculated as:

$$DPI^{avg} = \frac{DMISS}{INSTR} \tag{29}$$

In [6] it is proposed to divide the range between $CPI_{on}^{\min}$ and $CPI_{on}^{\max}$ uniformly and depending on the value of DPI^{avg} select one value from that range for CPI_{on}^{avg}. It is not directly explained the correspondence between the value of DPI^{avg} and selected CPI_{on}^{avg}. The extraction of DPI^{avg} in this implementation is done as follows:

$$\begin{aligned} \Delta DPI &= \max DPI - \min DPI \\ runDPI_i &= DPI_i - \min DPI \end{aligned} \tag{30}$$

$$\begin{aligned} procDPI &= \frac{runDPI}{\Delta DPI} \\ CPI_{on}^{avg} &= CPI_{on}^{\max} - procDPI \cdot (CPI_{on}^{\max} - CPI_{on}^{\min}) \end{aligned} \tag{31}$$

where $runDPI_i$ is DPI from the last quantum, *maxDPI* and *minDPI* are maximum and minimum values of DPI in last *N* quanta.

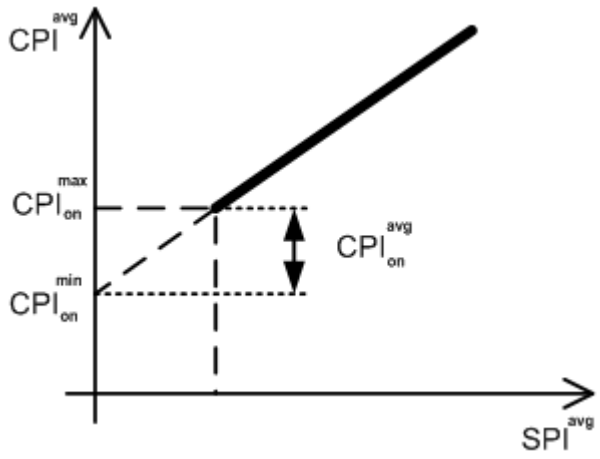

Figure 21 – Variables used for extraction of CPI_{on}^{avg}

Once, having CPI_{on}^{avg}, and therefore $T^{ON}_{f^{CPU}_{MAX}}$, it is possible to estimate β, and indirectly, $T_{f^{CPU}_{MAX}}$, for the i^{th} quantum.

$$\beta = \frac{T^{OFF}}{T^{ON}} = \frac{T^{quantum} - T^{ON}}{T^{ON}} = \frac{T^{quantum}}{T^{ON}} - 1 = \frac{CPI_i}{CPI_{on}^{avg}} - 1 \tag{32}$$

This value was the last one needed to estimate CPU frequency for the given performance loss.

VI. The implementation of the proposed DVS Algorithm

In the previous chapter was explained the theoretical part of the proposed algorithm. The topic of this chapter will be the implementation itself. As it was earlier said, the operating system that is used is Linux based. The algorithm itself is meant to be made as a periodic task. Linux operating system cannot offer any guarantee about the timings needed by the algorithm. Hence, Real Time Linux (RTLinux) is used as the operating system, and the DVS algorithm is implemented as one of real time modules that can be activated or deactivated at any time. RTLinux supports hard real-time (deterministic) operation through interrupt control between the hardware and the operating system. Interrupts needed for deterministic processing are processed by the real-time core, while other interrupts are forwarded to the non-real time operating system. The operating system (Linux) runs as a low priority thread. Figure 22 presents the layer architecture of RTLinux and its relationship with Linux

and real time modules. As it can be seen, the user's application are executed on the level of Linux operating system, and real time modules are executed on the same level, but with higher level of priority. RTLinux role is just to provide "real time" and to schedule given processes.

The period of collecting the data is a variable and the number of measurements used to calculate the beta, as well. Another problem was calculating the values from the algorithm, i.e. the problem was the process of real number division. The programming of real time module is done on very low level of abstraction, hence the mathematical functions for multiplication and division of real numbers cannot be used. All operations are done on integers. Due to this, all the input values are multiplied or divided with certain values in order to perform the calculus from the algorithm. Tables 7 and 8 show the values that were used in order to scale the algorithm's input values.

Once the algorithm was implemented and debugged, the next task was to provide means to measure application's execution time and energy consumption. Linux and RTLinux operating systems are based on the logic of pre-emption and priority levels of active application and the applications that are waiting in the queue. Each active process can be interrupted and be put in the waiting queue if a process with higher priority is activated. Due to this, it is important to distinguish the moments when the tested application is active from the moments when it is not. The only place where it can be done is the scheduler of the Linux operating system. Hence, in the part of kernel whose role is scheduling of active processes is implemented little filter. Whenever the tested application is activated, one of the XScale I/O pins is activated as well and set to logical 1. When the tested application is pre-empted by the high priority level process, this pin is set to logical 0. Measuring the time when the pin is active the application execution time is determined. Figure 23 shows a typical activity of application and its preemption by other active processes.

Variable	Constant used for scaling
CCNT	10
INSTR1	0.1
STALL	100
DMISS	10000

Table 7 – Variables that were scaled during the implementation

Variable	Number of decimal places obtained
CPI=(CCNT/INSTR1)	2
SPI=(STALL/INSTR1)	3
DPI=(DMISS/INSTR2)	6

Table 8 – The number of decimal places for application's variables obtained by scaling input data

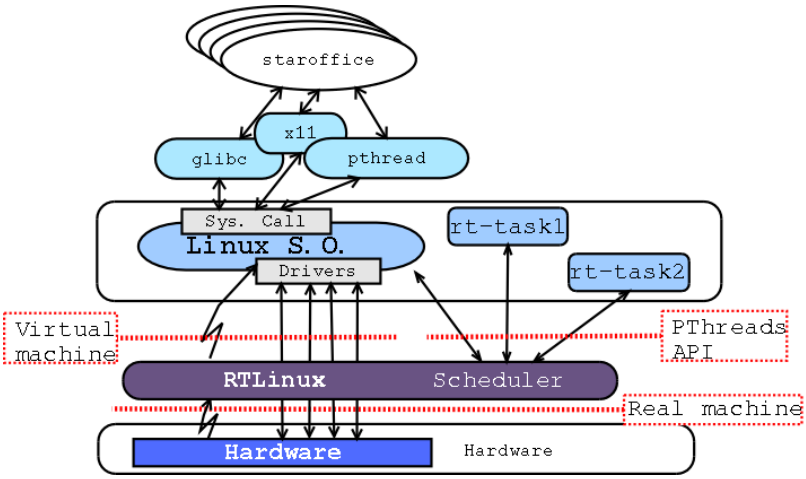

Figure 22 - Block diagram of RT Linux layer architecture

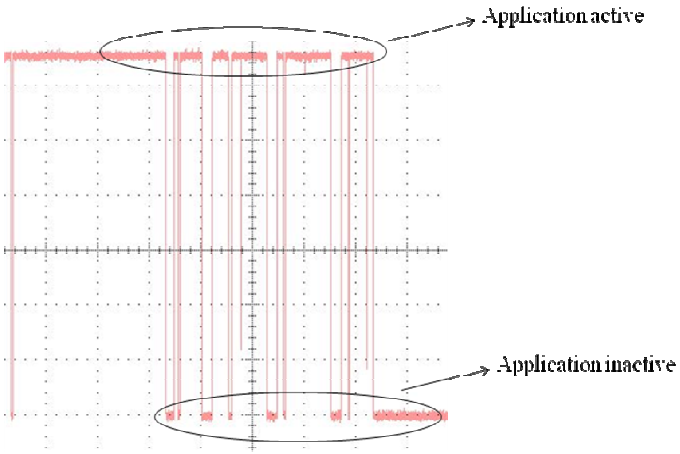

Figure 23 – Application's activity in time

This way of measurement is validated by using the *time* command (**time [***options***]** *command* **[***arguments...***]**). The *time* command runs the specified program *command* with the given arguments. When *command* finishes, *time* writes a message to standard output giving timing statistics about this program run. These statistics consist of: the elapsed real time between invocation and termination, the user CPU time and the system CPU time. The application's execution time is equal to the sum of user CPU time and system CPU time, and the same value was measured by the aforementioned method. Although the execution time could be measured using the *time* command, the proposed method has more profound (bigger) importance. As it was earlier said, the consumed energy is measured as well. It is measured using oscilloscope's voltage and current probes. Multiplying the values of CPU's voltage and current, we can know the power needed by the CPU's core. The multiplication of the power needed by the CPU and the logic signal, which is used to show application's activity, represents the energy consumed by the tested application.

It is in our interest to measure the time needed by the proposed algorithm and the energy needed for the voltage/frequency transitions. Hence, in order to measure the time, the same logic is applied as before, but this time the corresponding pin is set/reset whenever the algorithm is active/inactive, respectively, and the energy is calculated as before.

Using the proposed method and the control signals it is not necessary to know the priority level of tested application, as the control signal shows us when the tested application is active.

The last obstacle for the implementation of the algorithm is that there is a finite set of CPU frequencies. The algorithm calculates frequency of CPU that should be applied, but, because of the finite set of the CPU frequencies, the applied frequency is the one that is the closest one. Thus, there is an error in actual performance loss and this should be compensated. The compensation is done by recalculating the $PF_{DEMANDED} - PF_{for\_elected\_frequency}$, using the formula 20, and adding this value to $PF_{DEMANDED}$ before the next voltage/frequency change. For example, if the algorithm demands CPU frequency of 175 MHz, first is applied the frequency of 200 MHz, because it is the closest one, and then the one of 100 MHz , because the task will run faster than it is wanted so it is necessary to slow down the CPU. The algorithm would do it in the manner that the application lasts as if it was running 175 MHz all the time.

RESULTS

The experiments were conducted on three applications:

- gzip, application for file compression/decompression (a notepad file of 800kB was compressed)
- cjpeg, application for image compression (for example, from bmp format to jpg format)
- bfish, application for encryption of files

The control of execution time was first to be tested. The period of voltage/frequency changes was set to 20ms, and last 15 measurements of CPU's statistics were taken into account to estimate CPU's activity. In figure 24 the actual performance loss of the system is shown. It can be seen that is very close to the values that are demanded, all values are in ±5% of $PF_{DEMANDED}$. Figure 25 presents obtained energy saving and it can be noticed that energy savings are asymptotically drawing near certain value. The reason for this is that energy savings cannot be higher than in the case of the minimal CPU frequency. By increasing $PF_{DEMANDED}$, the average CPU frequency is getting closer to 100 MHz, so that energy savings are drawing near the maximum savings.

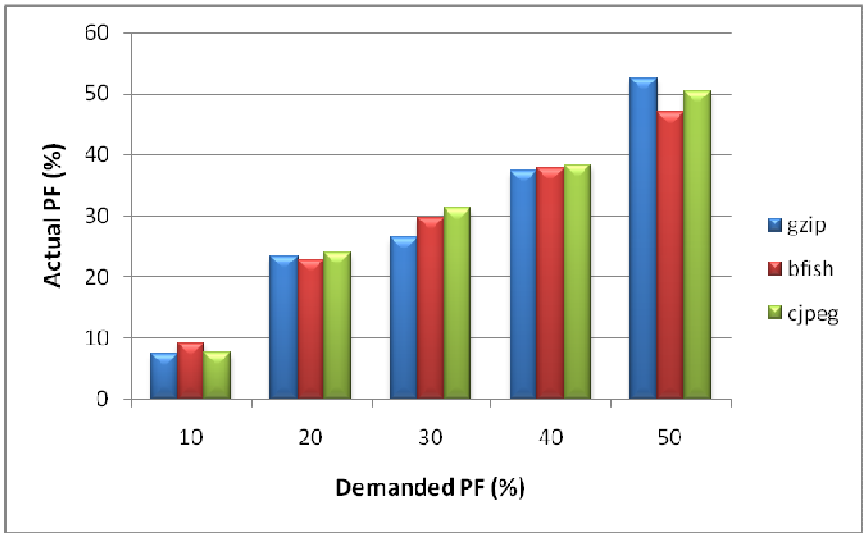

Figure 24 - Actual vs. demanded time performance loss

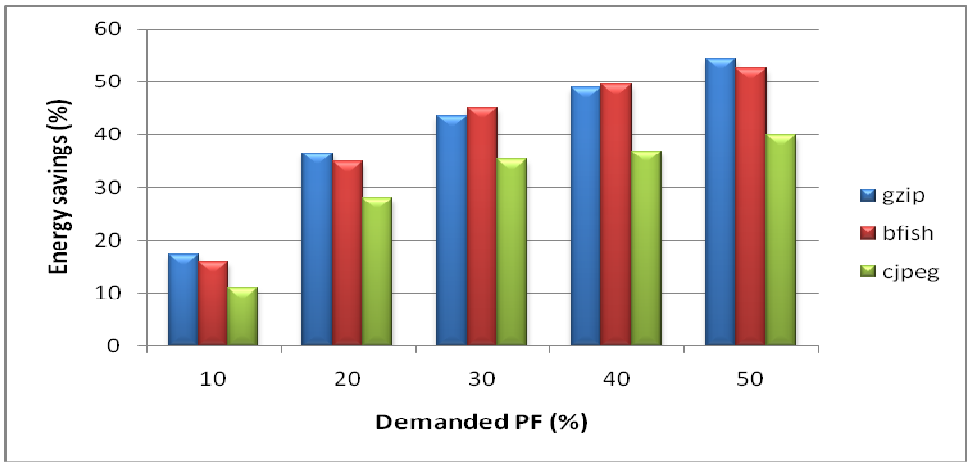

Figure 25 - Energy savings vs. demanded time performance loss

Figure 26 shows that without earlier mentioned compensation some values of Performance Loss cannot be correctly accomplished, for instance, when the demanded PF is 50% or 60%, the difference between the compensated algorithm and the algorithm without compensation is about 10%.

The time and the energy spent during transitions frequency/voltage should be negligible. In figures 27 and 28 the time of execution and the consumed energy of the proposed DVS algorithm are shown. The presented values are calculated regarding the time/energy of the tested application at the maximum speed. The maximum values are no higher than 5% of application's performance and energy, respectively. Each transition time is composed of the time needed for the voltage change and the time spent by PLL to lock to the new frequency. By measuring the times of the transitions it is determined that, approximately, 80% of the time needed by one transitions is spent by PLL. In order to see the influence of the power supply's dynamics we measured time and energy needed by the algorithm for two cases of its slew rate. In figures 29 and 30 is shown that because of the strong influence of PLL the dynamic of the power supply does not have great influence on the execution time and consumed energy of the algorithm.

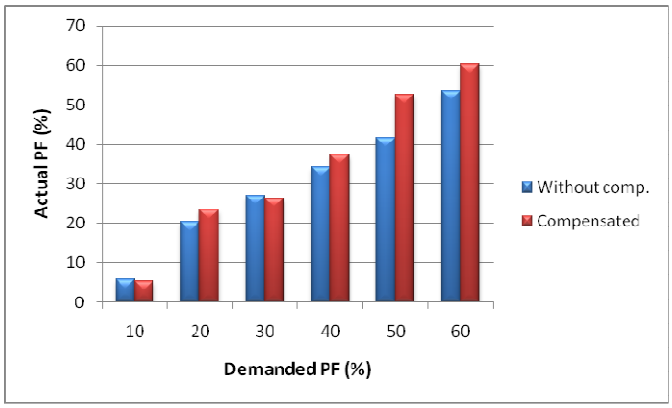

Figure 26 – Compensated algorithm vs. algorithm without compensation

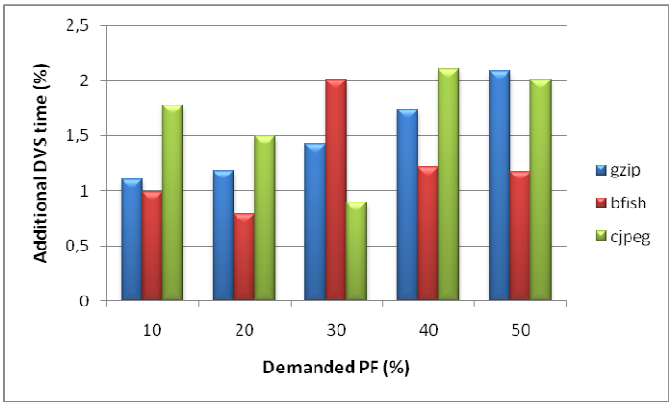

Figure 27 - Additional DVS time vs. demanded time performance loss

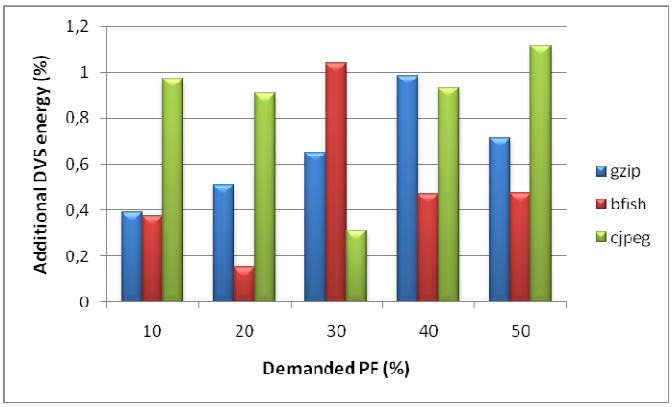

Figure 28 – Additional DVS energy vs. demanded time performance loss

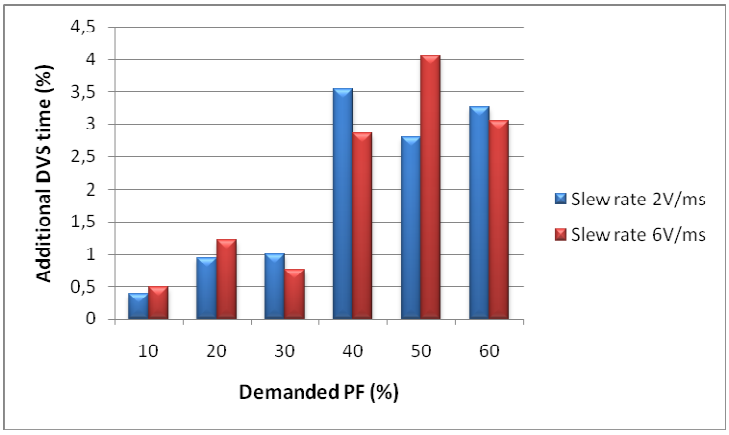

Figure 29 – Additional DVS time for different slew rates of the power supply

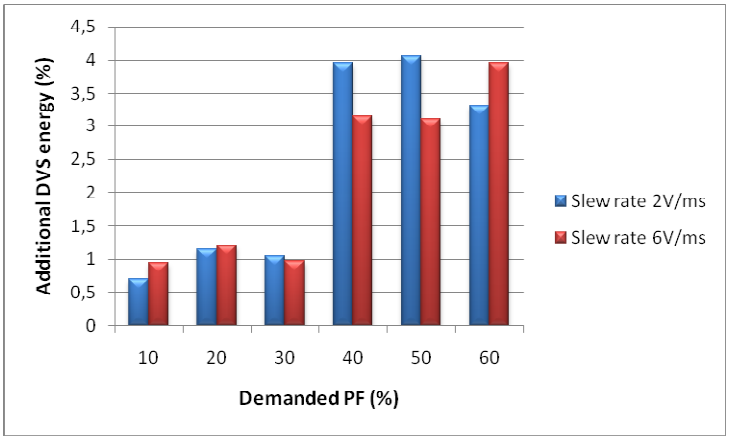

Figure 30 – Additional DVS energy for different slew rates of the power supply

VII. Problems of the proposed algorithm

As it was afore mentioned in the chapter about the theoretical background of algorithm, the time needed for off chip activities is estimated using a constant α. The value of the constant is evaluated by conducting a series of experiment in order to find the best value that would suite to whole range of PF_{WANTED}. Table 9 shows the values of α which were used for the tested applications.

Application	α	α' (1-α)
Gzip	0.90 for 800kB txt file 0.80 for 2MB txt file	0.10 for 800kB txt file 0.20 for 2MB txt file
Cjpeg	0.95	0.05
Bfish	0.97	0.03

Table 9 – Optimal value of a for tested applications

Hence, before applying the algorithm it is necessary to characterize the application and find the optimum α. Obviously, this is a drawback of the proposed algorithm. Even more, we found out that for the same application we need to adjust the value of α depending on the type of file that application uses as the object of its algorithm. For example, compression of a file filled with ASCII code (txt files) needs α of 0.95, and in the case of the same compression, but this time of an Acrobat Reader file (pdf), we could not find optimum value of α for all the range of $PF_{WANTED.}$ The results of the performance control for this case are presented in figure 31.

Similar problem we found out trying to compress a black and white photo to jpg format. The main cause of this problem can be in the statistics of the tested application. If we compare the statistics for gzip application in the case when we want to compress a txt file and when we want to compress a pdf file, we can see a significant difference between these two processes. In figures 32 and 33 are presented CPI vs. SPI diagrams and CPI vs. time for gzip in those two cases.

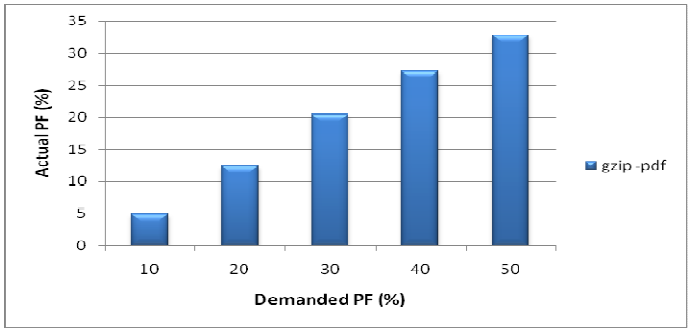

Figure 31 - Actual vs. demanded time performance loss for gzip applied to a pdf file

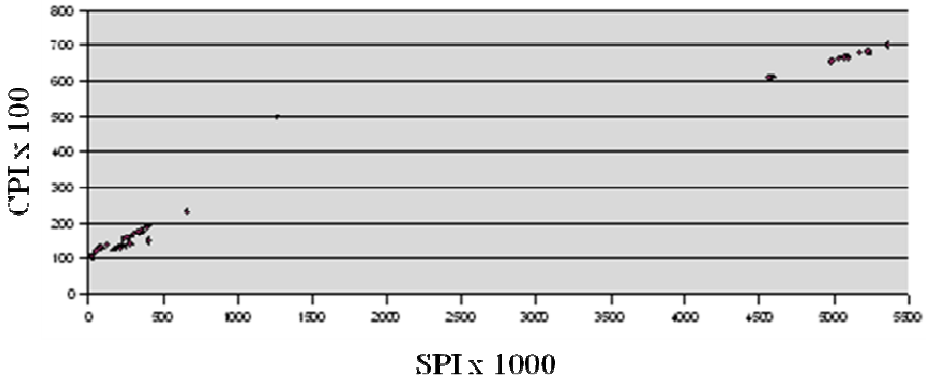

Figure 32 - Relationship between CPI and SPI when the algorithm does not work very well

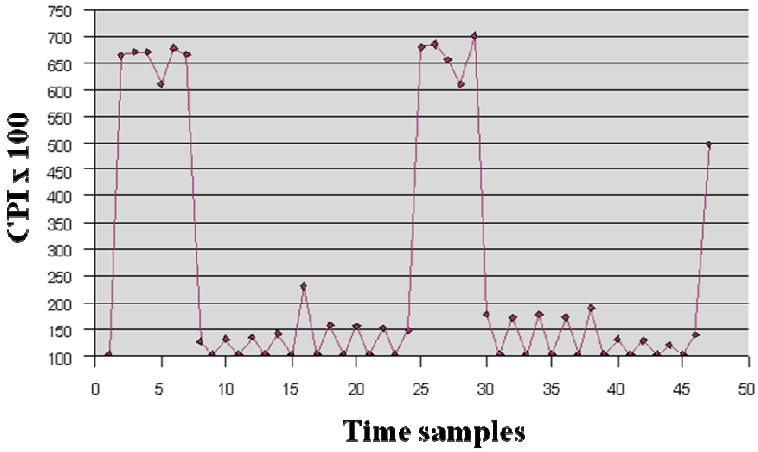

Figure 33 – CPI dynamic in time when the algorithm does not work very well

It looks like that CPU's activity during the compression of a pdf file is not high, so the algorithm uses high frequencies although relatively high PF is demanded. Because of low CPU activity, it was clear that the model for time off chip was inaccurate, that some part of the application's activity was not taken into account. Very similar difference we saw in the case of jpeg application, when we tried to compress a "regular" 16-bit photo on one side and a black and white photo on the other.

Due to this, we conducted a set of tests whose actions were strictly correlated with file management on the given flash disk. Figures 34 and 35 present CPI vs. SPI and CPI vs. time in the case of *copy* command. An 800 kB file was copied several times, one time after another and its statistics was taken.

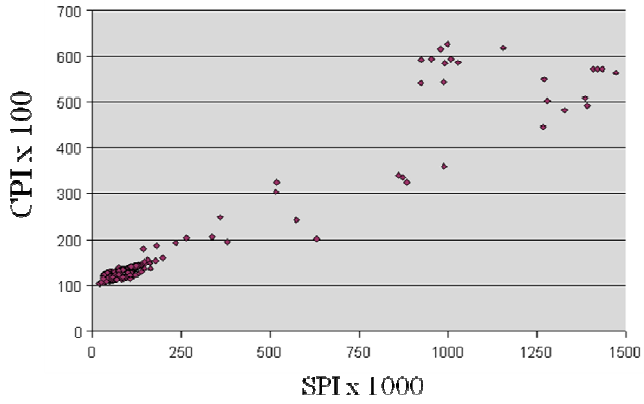

Figure 34 – CPI vs. SPI in the case of the "copy" command

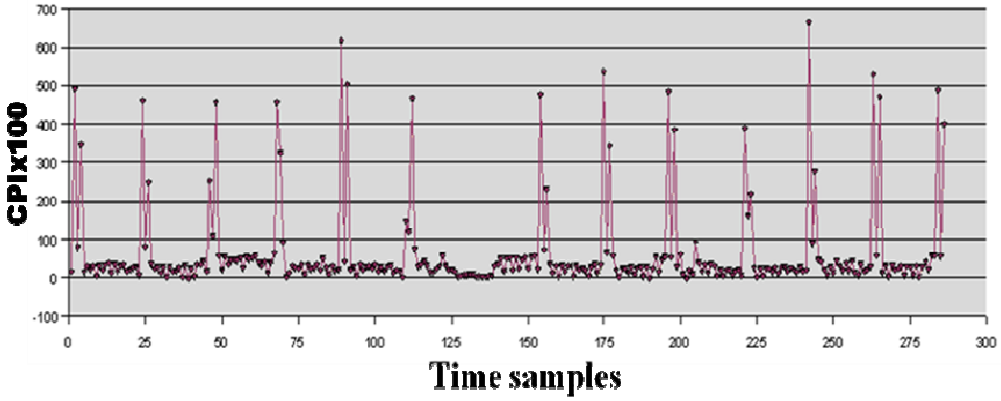

Figure 35 - CPI dynamic in the case of the "copy" command

As it can be seen, the activity of this command is very similar to the activity of the gzip application and a pdf file. This led to assumption that all the problems were because the algorithm did not model the time needed to read a source file, and later to write the results to an output file. Having in mind how the problem was found, trying to compress a file that is already compressed (there can be seen a significant difference in size of a pdf file made of any doc or txt file) or to compress a black and white photo, it looks like that the assumption could be right. It seems that the application is not high active because it does not have to do a lot of processing, hence all the activity is adjusted to read the input file and, then, to write to the output file (similar to the *copy* command). Anyway, this assumption true or not, it has to be tested more studiously with a good test bench.

Next challenge was to find the optimum period to take the CPU's statistics and change voltage and frequency. After doing several tests, we found that it is necessary to use a period that is at least one order of magnitude bigger than the time needed for voltage/frequency transition. For example, in the case of period 5 ms we needed to adjust the vale of α for several ranges of $PF_{DEMANDED.}$ In the case of bfish optimal α was changing its values from 0.75 for $PF_{DEMANDED}$ of 10% to 30% and then it was 0.85 for $PF_{DEMANDED}$ of 40% to 70%. Due to the definition of α, its value should be the same for each $PF_{DEMANDED}$. Hence, the period was being risen until it was possible to find unique α for a range of interest of $PF_{DEMANDED.}$ This could be explained that it is necessary to take a certain amount of information about CPU's activity in order to make good averaging of its activity level. If the period is small, the algorithm will try to adjust necessary voltage and frequency for each peak of activity or each fall of activity and it will make error, because the decision is made based on the statistics of time slice that has just passed. With a period sufficiently big, the peaks and falls of activity cannot influence heavily.

Conclusions

In this work, a DVS algorithm is presented. It is based on estimating the level of CPU's activity by measuring its statistics data (cycles per instruction, stall cycles per instruction and data cache misses per instruction) and trade off between time execution and energy needed by the application. Due to its nature to "predict" future by "looking" into past it is often addressed as a "history algorithm". The algorithms based on this idea are not suitable for DVS implementation on real time systems, because they cannot guarantee strict time execution that is needed for these systems. The proposed algorithm is implemented on a hardware platform with Intel XScale PXA255 microprocessor and Linux based operating system. The algorithm it self is made in such a way that system user can activate/deactivate it when he want it. This modularity is provided by implementing the algorithm as another system module.

The implemented algorithm is aware of the limits of the hardware platform, it uses only four frequencies for the system clock and four voltages for the CPU's core and it limits the maximal speed of change for core voltage reference in order to work properly after the voltage change [32].

The proposed DVS algorithm is tested with several applications that are often used on microprocessor platforms. The tested applications are used to compress files (gzip), compress photos (cjpeg) or to encrypt a file (bfish). Applying the proposed DVS algorithm with mentioned applications, we came to next conclusions:

- It is possible to achieve up to 50% of energy savings with 50% of time performance loss.
- The control of desired execution time is relatively good; the error is not bigger than ±5% of time needed at maximum frequency.
- The additional energy and time needed by the algorithm and voltage/frequency transitions are below 2% of the energy and time spent by the tested application at the maximum frequency.
- It is necessary to implement the algorithm with the closed loop, where the present PF is the feedback signal. If not, due to the limited number of frequencies of system clock the error in performance can be very big.
- The period of voltage and frequency changes needs to be sufficiently big to estimate average activity of the microprocessor. In the case of small time periods the peaks of applications activity are very dominant and can lead algorithm to bad performance. When the period is big enough these peaks are filtered and the average activity is better estimated.

- Due to strong dependency of the algorithm on the measured statistics, it is necessary to characterize the applications that will be used within the DVS system. The purpose of the application's characterization is adjustment of the model that is used in the algorithm. The adjustment of the model is one of the drawbacks of this algorithm.
- Another problem is bad behavior of the algorithm when there is no high CPU activity. This problem is yet to be solved.

We tried to see if there was relationship between the energy savings and power supply's dynamic. The algorithm was tested in two cases. First, when the slew rate of the power supply was 2V/ms and then with the slew rate of 6V/ms. The energy spent by the algorithm in both cases was practically the same, so we did not see any difference between those two cases of slew rate. Nevertheless, this was expected having in mind the period of voltage/frequency changes and comparing the dynamic of PLL that is inside the CPU with the dynamic of the power supply. Due to large time intervals needed by the PLL to lock it self at certain frequency, we came to conclusion that for this implementation of DVS idea fast power supply is not of crucial significance. What it is important is a good theoretical model of the system.

Future work

The algorithm should be resistive on the application that it is tested and the type of data that is processed. Due to this, it is necessary to improve the algorithm to make the estimation of the value of α and to make better model of off-chip time.

The proposed algorithm can be related to the, so-called, "history" algorithms. It cannot be used in real time system where is necessary to obey time restrictions. One possibility for future work can be an algorithm for real time systems on the given hardware platform.

We conducted some experiments with FPGA chips and selected them as a possible platform for DVS idea. Figures 36 and 37 show maximal frequency of FPGA's clock for different core's voltages and possible energy savings if this technique is used.

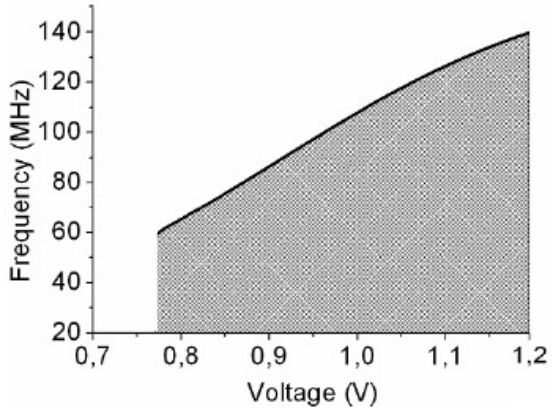

Figure 36 – Maximal frequency for different core's voltages

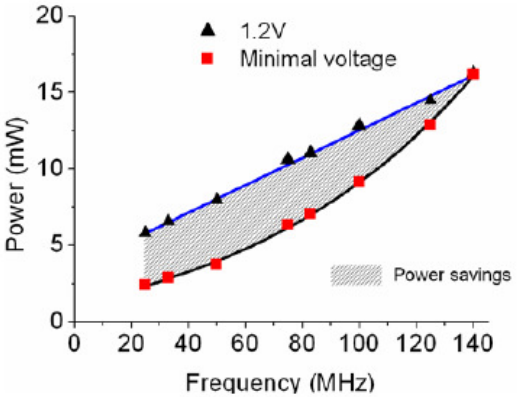

Figure 37 – Possible energy savings if DVS is applied

References

[1] http://www.transmeta.com/pdfs/presentations/060517_mpf_spring_2006_tmta_longrun2.pdf

[2] http://support.intel.com/support/processors/mobile/pentiumiii/sb/CS-007509.htm

[3] http://www.intel.com/support/processors/mobile/pentiumiii/sb/cs-007522.htm

[4] AMD PowerNow!™ Technology, Dynamically Manages Power and Performance, Informational White Paper

[5] Choi, K.; Soma, R.; Pedram, M.: Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Tradeoff Based on the Ratio of Off-Chip Access to On-Chip Computation Times. IEEE transactions on computer-aided design of integrated circuits and systems, vol. 24, No. 1, Jan. 2005

[6] Choi, K.; Soma, R.; Pedram, M: Dynamic Voltage and Frequency Scaling based on workload Decomposition. Proceedings of the 2004 International Symposium on Low Power Electronics and Design, ISPLED, 2004

[7] Kawaguchi, H.;Shin Y.;Sakurai T.: uITRON-LP: Power-Conscious Real-Time OS Based on Cooperative Voltage Scaling for Multimedia Applications. IEEE transactions on multimedia, Vol. 7, No. 1, February 2005

[8] Saewong S.; Rajkumar R.: Practical Voltage-Scaling for Fixed-Priority RT-Systems. Proc.of the 9th IEEE Real-Time and embedded Techn. and Applications Symposium (RTAS'03) 2003 IEEE, 2003

[9] Son D.;Yu C.; Kim H.: Dynamic Voltage Scaling on MPEG Decoding, ICPADS, 2001

[10] A Dynamic Voltage Scaled Microprocessor System Thomas D. Burd, Student Member, IEEE, Trevor A. Pering, Anthony J. Stratakos, and Robert W. Brodersen, Fellow, IEEE

[11] TM5800 Version 2.1 Data Book

[12] TM5500/TM5800 System Design Guide

[13] The Technology Behind Crusoe™ Processors Low-power x86-Compatible Processors Implemented with Code Morphing™ Software Alexander Klaiber Transmeta Corporation January 2000

[14] Data sheets, Designers guieds for AMD K6

[15] Datasheets, Developer's guides of the Intel's 80200 processor

[16] A. Soto, P. Alou, J.A.Cobos, J,Uceda, "The future DC-DC converter as an enabler of low energy consumption systems with dynamic voltage scaling," IECON 02, vol. 4, pp. 3244 – 3249, November 2002

[17] A. Soto, A. de Castro, P. Alou, J.A. Cobos, J. Uceda and A. Lotfi , "Analysis of the Buck Converter for Scaling the Supply Voltage of Digital Circuits," Applied Power Electronics Conference and Exposition, 2003. APEC '03. Eighteenth Annual IEEE, vol.2 , Feb. 2003.

[18] T.Kuroda, K.Suzuki, S.Mita, T.Fujita, F.Yamane, F.Sano, A.Chiba, Y.Watanabe, K. Matsuda, T. Maeda, T.Sakurai and .Furuyama, "Variable supply-voltage scheme for low-power high-speed CMOS digital design," Solid-State Circuits, IEEE Journal, vol. 33, Issue 3, March 1998

[19] O.Trescases and W.T.Ng, "Variable output, softswitching DC/DC converter for VLSI dynamic voltage scaling power supply applications," Power Electronics Specialists Conference, 2004. PESC04. 2004 IEEE 35th Annual, vol. 6, pp. 20-25, June 2004

[20] S.Dhar, D. Maksimovic, "Switching Regulator with Dynamically Adjustable Supply Voltage for Low Power VLSI," Industrial Electronics Con. IECON'01, 2001

[21] A. Soto, P. Alou and J.A.Cobos, "Design Methodology for Dynamic Voltage Scaling in the Buck Converter," Universidad Politécnica de Madrid, División de Ingeniería Electrónica, 2005.

[22] http://www.arcom.com/devkit-wince-viper.htm

[23] Gruian, F.: Energy-Efficient Scheduling for Hard Real-Time Applications on Dynamic Voltage Supply Processors. Euro. Summer School on Embedded Systems, Sweden, 2003

[24] M.Weiser, B. Welch, A. Demers and S.Shenker, „Scheduling for reduced CPU energy", Proc. 1st USENIX Symp. On Operating Systems Design and Implementation, pp. 13-23, Nov. 1994.

[25] K. Govil, E. Chan, and H. Wasserman, "Comparing algorithms for dynamic speed-setting of a low-power CPU", Proc. ACM Int'l Conf. on Mobile Computing and Networking, pp. 13--25, Nov. 1995.

[26] Choi, K.; Soma, R.; Pedram, M.: Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Tradeoff Based on the Ratio of Off-Chip Access to On-Chip Computation Times. IEEE transactions on computer-aided design of integrated circuits and systems, vol. 24, No. 1, Jan. 2005

[27] Choi, K.; Soma, R.; Pedram, M: Dynamic Voltage and Frequency Scaling based on workload Decomposition. Proceedings of the 2004 International Symposium on Low Power Electronics and Design, ISPLED, 2004

[28] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. J. ACM 20, 1 (Jan. 1973), 46–61.

[29]] P. Pillai, K. G. Shin, " Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems" Proc. of the 18th ACM Symp. on Operating Systems Principles, 2001

[30] D. Son, C. Yu, H. Kim, "Dynamic Voltage Scaling on MPEG Decoding" International Conference of Parallel and Distributed System (ICPADS), June 2001

[31] http://www.arcom.com/devkit-wince-viper.htm

[32] Viper Intel PXA255 XScale RISC based PC/104 single board computer, technical manual

[33] http://www.maxim-ic.com/quick_view2.cfm/qv_pk/3404

[34] Computer Architecture: A Quantitative Approach, Third Edition, John L. Hennessy , David A. Patterson