Static WCET Analysis Based Compiler-Directed DVS Energy Optimization in Real-Time Applications*

Yi Huizhan, Chen Juan, and Yang Xuejun

Section 620, School of Computer, National University of Defense Technology, Changsha, 410073, Hunan, P.R. China
{huizhanyi, juanchen, xjyang}@nudt.edu.cn

Abstract. Compiler-directed dynamic voltage scaling (DVS) is one of the effective low-power techniques for real-time applications. Using the technique, compiler inserts voltage scaling points into a real-time application, and supply voltage and clock frequency are adjusted to the relationship between the remaining time and the remaining workload at each voltage scaling point. In this paper, based on the *WCET* (the worst case execution time) analysis tool *HEPTANE* and the performance/power simulator *Sim-Panalyzer*, we present a DVS-enabled simulation environment *RTLPower* (**R**eal-**T**ime **L**ow **P**ower), which integrates static *WCET* estimation, performance/power simulation, automatically inserting the DVS code into a real-time application, and profile-guided energy optimization. By simulations of some benchmark applications, we prove that the DVS technique and the profile-guided optimization technique significantly reduce energy consumption.

Keywords: Real-time, Low-power, WCET, Compiler.

1 Introduction

In the recent years, embedded systems for mobile computing, such as mobile phone and PDA, are developing rapidly, and a crucial parameter of mobile systems is the continued time of energy supply. Although the performance in the integrated circuits (ICs) has been increasing rapidly in recent years [1], battery techniques are developed very slowly [2] and it is of significant importance for battery-powered mobile systems to utilize more effective low-power techniques.

Many novel low-power techniques in circuit, logic, architecture and software levels, in order of increasing abstraction, have been proposed to reduce energy consumption. Dynamic voltage scaling (DVS) [3], [4] is one of the low-power techniques in architecture level, and it is widely used in embedded systems for mobile computing and desktop systems. Real-time dynamic voltage scaling dynamically reduces supply voltage to the lowest possible extent that ensures a proper operation when the required performance is lower than the maximum performance. Since the dynamic energy consumption, the dominant energy consumption in ICs, is in direct

* Supported by the National High Technology Development 863 Program of China under Grant No. 2004AA1Z2210 and Server OS Kernel under Grant No. 2002AA1Z2101.

C. Jesshope and C. Egan (Eds.): ACSAC 2006, LNCS 4186, pp. 123 – 136, 2006.

proportion to the square of supply voltage *V*, it is possible for DVS to significantly reduce energy consumption.

The voltage scheduling in a single task called an intra-task dynamic voltage scaling (IntraDVS) [7] is proposed. IntraDVS assisted by compiler automatically inserts voltage scaling points into a real-time task and divides the task into some execution sections, and then supply voltage is adjusted to the relationship between the remaining time and the remaining workload.

It is crucial for IntraDVS to properly place voltage scaling points in a real-time application, and the configuration of voltage scaling points significantly affects energy consumption. A good configuration could save more energy; however, due to voltage scaling overhead, the improper one could waste much energy. For the past few years, much work has been published on compiler-directed real-time dynamic voltage scaling [5], [6], [7], [8], [9], [10], [11], [12], [13], and the algorithms have utilized two kinds of configurations of voltage scaling points. The first is to make use of fixed-length voltage scaling sections, the whole execution of a task is divided into some equal subintervals and the voltage adjustment is made at the beginning of each subinterval [6] [11]. The second is a heuristic method, the condition and loop structure in real-time applications often bring about the workload variation and energy consumption can be reduced enormously if voltage scaling points are put at the end of the structures [7] [13]. Yi, et al proved that the heuristic configuration is the optimal one when not considering the voltage scaling overhead [14]. At the same time they presented a profile-guided optimizing configuration methodology, and using some synthetic applications, they proved that the methodology significantly reduces energy consumption. But they have not explained how to realize the method, and no experimental results of real benchmark applications are given. Another problem of the past works is not integrating with the WCET analysis tightly, but for real-time applications, it is a key to give the time estimate method in detail.

In this paper, based on the *WCET* (the worst case execution time) analysis tool *HEPTANE* and the performance/power simulator *Sim-Panalyzer*, we present a DVS-enabled simulation environment *RTLPower*, which integrates static *WCET* estimation, performance/power simulation, automatically inserting the DVS code into a real application, and profile-guided energy optimization. By simulations of some real benchmark applications, we prove that the DVS technique and the profile-guided optimization technique significantly reduce energy consumption.

The rest of this paper is organized as follows. In Section 2, we list the related terms of compiler-directed dynamic voltage scaling. In Section 3, we give the inserting method of DVS code. In Section 4, we present the profile-guided energy optimization method. In Section 5, we show by experiments that the DVS technique and the profile-guided optimization technique significantly reduce energy consumption. Finally, we give the conclusions.

2 Related Terms

A real-time task has strict timing constraint and must finish before its deadline (*d*), missing the deadline might lead to a catastrophic result. Real-time dynamic voltage scaling guarantees a correct operation of a real-time task and dynamically reduces

supply voltage and clock frequency to the lowest possible extent in the execution course. Therefore, for real-time applications, the worst-case execution time (*wcet*) or the worst-case execution cycle (*wcec*) must be estimated in advance [19] to ensure that the timing constraint is met, that is, the worst-case execution time must be less than or equal to the deadline. If the *wcet* is less than the deadline, we can proportionally reduce clock frequency beforehand. Consequently, the *wcet* is equal to the deadline *d* and the obtained initial frequency is f_{static}, that is, $d=wcec/f_{static}$. This is the starting point of dynamic voltage scaling in this paper. Current DVS-enabled systems only can change the clock frequency on some discrete levels [15], [16], [17], and therefore we assume that the clock frequency can change on some discrete levels between consecutive interval $[f_{min}, f_{max}]$.

IntraDVS divides the whole execution cycle of a task into n sections, and the worst-case execution cycle and the actual execution cycle of each section are denoted by wc_i and ac_i for $i = 1,...,n$, respectively. It is obvious that $0 \le ac_i \le wc_i$ for $i = 1,...,n$, and $wcec = \sum_{l=1}^{n} wc_l$. The reduced worst-case execution cycle of the *i*th point is denoted by $rwec_i$ for $i = 1,...,n+1$, and we have $rwec_i = \sum_{l=i}^{n} wc_l$ for $i = 1,...,n$, $rwec_{n+1} = 0$.

At the beginning of each section, supply voltage (V_i for $i = 1,...,n$) and clock frequency (f_i for $i = 1,...,n$) are adjusted to the relationship between the remaining time and the remaining workload, and the lowest supply voltage and clock frequency are utilized within timing constraint. The proportional voltage scaling sets the frequency of the *i*th section to

$$f_i = rwec_i / (d - \sum_{l=1}^{i-1} t_l)$$

where t_l denotes the actual execution time of the *l*th section. In the above formula, the new clock frequency at the beginning of the *i*th section is set to the quotient of the reduced worst-case execution cycle divided by the reduced time, which can guarantee that the task can finish before its deadline at any time.

The formula $f \propto (V - V_T)^2 / V$ defines the relationship between clock frequency and supply voltage of CMOS, where V_T denotes the threshold voltage of CMOS.
The execution time t_i of each section can be computed by

$$t_i = ac_i / f_i$$

Finally, dynamic voltage scaling have some energy overhead and time overhead, which are closely related to the initial voltage V_{DD1}, the final voltage V_{DD2}, and the switch capacitance *C*. Burd, et al [18] present the formula of energy overhead

$$E = (1-\eta) \cdot C \cdot \left| V_{DD2}^2 - V_{DD1}^2 \right|$$

and the formula of time overhead

$$t_{TRAN} = \frac{2 \cdot C}{I_{\max}} \cdot \left| V_{DD2} - V_{DD1} \right|$$

In this paper, we let $\eta = 0.9$ (the typical value) and $C = 5pF$. The time overhead is fixed as 200 cycles for 100Mhz frequency variation.

3 Inserting Method of DVS Code

Based on the *WCET* analysis tool *HEPTANE* [20], we present an automatically inserting method of DVS code. For a real application program, it includes condition structures, loop structures, and function calls, besides the sequential codes. Our method can insert into any location of an application program. At each voltage scaling point, we need three parameters: the reduced worst case execution cycle ($rwec_i$), the deadline (d), and the current time (ct). The deadline is defined before hand, and the current time can be obtained dynamically from the simulation system. The modified simulation system *Sim-Panalyzer* can accumulate the actual execution time, and convey the time information to real-time applications by some predefined memory port. Therefore, if $rwec_i$ is known, we can set the supply voltage and clock frequency of each voltage scaling point. Furthermore, in order to make it possible to optimize the insertion of voltage scaling points, we make each point executed by a prediction *insert_or_not[i]*. Therefore, at each point, the DVS pseudo-code is illustrated at Fig. 1. The function *getcurrenttime* obtains the current actual execution time from the simulation environment. The function *setnewfrequency* sets new system execution frequency, and based on the remaining time and the remaining workload, the function computes new frequency and sets the nearest discrete voltage/frequency level that guarantees the real-time execution. Both functions are realized by embedded assemble language, which is supported by GCC compiler. Based on the different cases, *computecurrentRWEC* can correctly give the reduced worst case execution cycle.

```
1 if (insert_or_not[i]) {
2     getcurrenttime(ct);
3     computecurrentRWEC(rwec_i);
4     nf = rwec_i / (d-ct-overhead);
5     setnewfrequency(nf);
6 }
```

Fig. 1. The pseudo-code of DVS at each point

The time estimation process of the *WCET* analysis tool *HEPTANE* is as follows:

1. Based on the source code of an application, a syntax tree is produced, which corresponds to the source code structure.
2. From the syntax tree, a context tree is formed, which corresponds to the execution process of the application. At the same time, some labels are inserted in the syntax tree, which are used to mark the basic block (no branch structure). The resultant syntax tree is used to output the modified source code.
3. The modified source code is compiled using GCC compiler, and the assemble file and binary file are produced. Using the specific architecture information (such as cache size, pipeline stage), the worst case execution time of each basic block is estimated.
4. Using the context tree and the time information of basic blocks, the worst-case execution time of the whole application is accumulated by depth-firstly traversing the context tree.

Based on the time estimation process of *HEPTANE*, we select inserting the DVS code in the syntax tree of the source code. By traversing the syntax tree, we mark the location for all the DVS points. When outputting the modified source code, the DVS code is inserted into the source code automatically. Therefore, the final time estimation includes the execution time of the DVS code (not DVS overhead), and the safe real-time DVS program is produced.

Next, we present how to correctly get the value of $rwec_i$ for the different cases. For the code without loops and function calls, it is simple to make use of *HEPTANE* to estimate the worst case execution cycle of each point. For the voltage scaling points inserted into loops, the $rwec_i$ of each iteration is different. Similarly, for the voltage scaling point inserted into function calls, the different call sites of the function have the different $rwec_i$. Since it is possible for our inserting method to insert a point into any place, we must solve the problem due to loops and function calls.

```
Loop1::for(i1...)                    __index1…__indexN = 0
  Loop2::for(i2...)                  Loop1::for(i1...)
                                       Loop2::for(i2...)
    LoopN::for(iN...)
        Votage scaling point             LoopN::for(iN...)
    LoopN end                                Votage scaling point
                                         __indexN++
  Loop2 end                              LoopN end
Loop1 end                                __indexN =0
                                       Loop2 end
                                       __index2 =0
                                       __index1++
                                     Loop1 end
                                     __index1 = 0
```

Fig. 2. The pseudo-code of instrumented code for loops

3.1 Compute the $rwec_i$ of Voltage Scaling Points Inserted into Loops

For the voltage scaling points inserted into loops, the $rwec_i$ of each iteration has the different value, which is closely related to the specific iteration. We give the $rwec_i$ by the parametric method:

$$rwec_i = rwec_{base_i} + \sum_{j=0}^{n} (loop^j_{\max\_iteration_i} - loop^j_{cur\_iteration_i} - 1) \cdot wcec^j_{loop_i}$$

where $rwec_{base_i}$ is the reduced worst case execution cycle for last iterations of all loop levels including the voltage scaling point, $wcec^j_{loop_i}$ is the worst case execution cycle of the *j*th loop level, $loop^j_{\max\_iteration_i}$ is the maximum iteration number of the *j*th loop level, $loop^j_{cur\_iteration_i}$ is the current iteration number of the *j*th loop level. Here, we specify the compute method of $rwec_i$ when there are no function calls.

Using the *HEPTANE* tool, we can obtain $loop^{j}_{\max\_iteration_i}$ from the loop annotation, whereas $loop^{j}_{cur\_iteration_i}$ need add some instrumented code. Based on the syntax tree of the source code, we insert the instrumented code to get $loop^{j}_{cur\_iteration_i}$, as is shown in Fig. 2. At the beginning of the loop, the indexes of the corresponding loop levels are initiated to zero. When getting into a more deep loop level, the corresponding index is incremented; on the contrary, when getting out of a loop level, the index is cleared to zero. The instrumented code is directly inserted into the syntax tree, and when outputting the modified source code, the result includes the instrumented code.

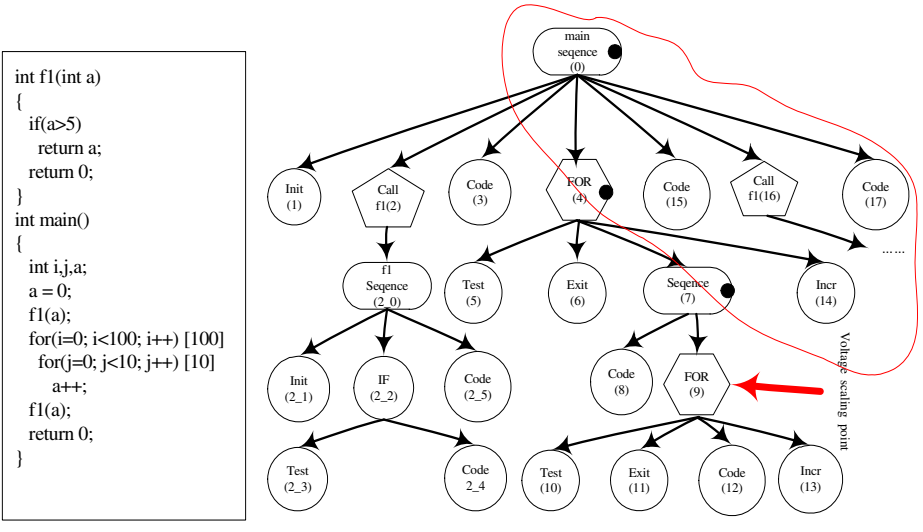

Fig. 3. A source code and the corresponding context tree for time estimation

We can estimate $wcec^{j}_{loop_i}$ directly using *HEPTANE* tool, but cannot directly get $rwec_{base_i}$ from *HEPTANE*. In order to compute $rwec_{base_i}$, we modified the *HEPTANE* tool, and estimate time by pruning the context tree of *HEPTANE*. The source code and the corresponding context tree for time estimation are shown in Fig. 3. The source code is a typical program except including some annotations of the maximum number of loop iterations. The context tree is an expanded syntax tree, and it consists of all execution instances of the functions. Suppose For(9) is selected as an voltage scaling point, then only the nodes surrounded by the free curve have contribution to the $rwec_{base_i}$, the nodes marked by the black dots only need to estimate the time of their partial sub-nodes. From the voltage scaling point, we search the parent node, prune the sibling node before current node, and maintain the current node and the sibling node after current node. For loop node, only a single iteration is considered. Finally, we estimate the worst case execution cycle of the reduced context tree from bottom to top, which is equal to $rwec_{base_i}$.

3.2 Compute the $rwec_i$ of Voltage Scaling Points Inserted into Function Calls

If the voltage scaling points are inserted into function calls, the $rwec_i$ is different for each instance of a function call. We add the hint information of function calls to estimate the $rwec_i$:

$$rwec_i = rwec_{func_i} + rwec_{loop_i}$$

$$rwec_{loop_i} = rwec_{base_i} + \sum_{j=0}^{n} (loop^j_{\max\_iteration_i} - loop^j_{cur\_iteration_i} - 1) \cdot wcec^j_{loop_i}$$

where $wcec^j_{loop_i}$, $loop^j_{\max\_iteration_i}$, and $loop^j_{cur\_iteration_i}$ have the same meanings as before, $rwec_{base_i}$ is the reduced worst case execution cycle between current voltage scaling point and the end of the function, $rwec_{func_i}$ is the reduced worst case execution cycle of the end of current instance of the function call. For the different instances of function call, $rwec_{base_i}$ could have the different value, and we simply use the maximum $rwec_{base_i}$ for all instances. Using the $rwec_{func_i}$, we can differ one instance from the others.

```
f1()                          f1(float rwec)
{                             {
   voltage scaling point         voltage scaling point
}                             }
f2()                          f2(float rwec)
{                             {
   f1()                          rwec1 = rwec+rwec_a;
   f1()                          f1(rwec1)
}                                rwec2 = rwec+rwec_b;
                                 f1(rwec2)
                              }
```

Fig. 4. The pseudo-code of instrumented code for function calls

As before, we need insert the instrumented code, and an example is shown in Fig. 4. The function *f1* includes a voltage scaling point, we add a parameter for the function *f1*, which represent the $rwec_{func_i}$ for the function *f1*. When the function *f1* is called, we can know the reduced worst case execution cycle at the end of the current instance of the function *f1*. Similarly, the function *f2* includes the call instance of the function *f1*, and then it also needs an additional parameter to represent the reduced worst case execution cycle at the end of the instance of the function *f2*. At the same time, we make use of *HEPTANE* to estimate the worst case execution cycle *rwec_a* between the end of the first instance of the function *f1* and the end of the function *f2*, and the worst case execution cycle *rwec_b* between the end of the second instance of the function *f1* and the end of the function *f2*. As a result, we can compute the $rwec_{func_i}$ for two instances of the function *f1*, which are transferred to the voltage

scaling point by the function parameter. Besides the voltage scaling points, some other points such as *rwec_a* and *rwec_b* also need estimate the reduced worst case execution cycle, and we call them assistant voltage scaling points.

For each voltage scaling point, we find out the function including the point in the syntax tree. Then, we search for the syntax tree and find out all the call sites of the function. The call sites are the assistant voltage scaling points. Combining all the voltage scaling points with the assistant voltage scaling points, we continue to find out more assistant voltage scaling points till the number of the voltage scaling points is not changed. For the different kinds of voltage scaling points, we insert the corresponding code and correctly compute the $rwec_i$. The pseudo-code algorithm of searching for voltage scaling points is shown in Fig. 5.

```
Input:
     list_dvspoint represents a list of all initial voltage scaling
     points (the end of the uncertain loop and the beginning of each
     condition path )
1 while(the size of list_dvspoint is changed)
2       while( list_dvspoint is not empty)
3           get a dvs point
4           search for the function f including the dvs point
5           search for all the call sites of f, insert into a call site list
            list_callpoint
6       combine list_callpoint with list_dvspoint, get an updated
        list_dvspoint
7 for(each point in the list_dvspoint)
8       if(dvs point)
9           output the RWEC computing code
            and the voltage scaling code
10      else if(assistant dvs point)
11          output the RWEC computing code, add the function
            parameter and the parameter of function call
```

Fig. 5. The pseudo-code algorithm of searching for voltage scaling points

4 Profile-Guided Energy Optimization

When not considering the voltage scaling overhead, the optimal configuration minimizing the energy consumption inserts voltage scaling points at the end of the uncertain loop (for example, "while" in C language) and the beginning of each condition path (for example, if-then-else and "switch" in C language). We realize the inserting method, search for the syntax tree, and insert voltage scaling points into the end of each uncertain loop and the beginning of each path of the condition structure.

When considering the voltage scaling overhead, the inserting method are not the optimal. Yi, et al [14] have presented an analytical energy model, and based on the energy model, they give an optimizing method, which deleted overmany voltage

scaling points from the initial set of voltage scaling points. The optimizing method considers each time voltage adjustment as a voltage scaling point, and attempts to maintain the optimal voltage adjustment. For a real application program, voltage scaling points can be inserted into any place, each voltage scaling points can correspond to multiple instances. For example, as shown in Fig. 2, a voltage scaling point is inserted into a loop, and any iteration of the loop has made voltage adjustment. For the voltage scaling point included in Fig. 4, each instance of the function *f1* corresponds to one voltage adjustment. It is not simple problem to delete voltage adjustment of an application.

Generally speaking, a voltage scaling point corresponds to an inserting location. For example, for the voltage scaling point included by *f1* in Fig. 4, we consider it as a voltage scaling point. When we delete the voltage scaling point, we really delete two times voltage adjustment corresponding to two instance of the function *f1*. It is obvious that the voltage scaling point definition can lead to the ineffective optimization, and the main problem comes from the voltage scaling points inserted into loops. For example, for the voltage scaling point in Fig. 2, it is possible that it corresponds to a large number of voltage adjustment, and as a result, its deletion leads to ineffective voltage scaling placement. Therefore, we need give special meaning for the points inserted into loops.

```
1 if (insert_or_not[i] && indexj mod stride == 0) {
2     getcurrenttime(ct);
3     computecurrentRWEC(rwec_i);
4     nf = rwec_i / (d-ct-overhead);
5     setnewfrequency(nf);
6 }
```

Fig. 6. The modified pseudo-code of DVS inserted into loops

We consider the point inserted into loops as multiple voltage scaling points and need to be deleted in some sequence. Taking into account a modified DVS pseudo-code as shown in Fig. 6, we add a prediction (*indexj* mod *stride*), where *indexj* is the index of the *j*th loop level, *stride* is the stride length, and mod represents the modulus operator. We can delete a voltage scaling point by clearing *insert_or_not[i]* to zero. For the voltage scaling points inserted into loops, we also can delete some voltage adjustment by setting *indexj* and *stride* to the different value. Therefore, we divide the original optimizing methods [14] into two steps: optimizing the points inserted into loops and globally optimizing the points inserted into the application. At the first step, we delete the voltage adjustment by the order that firstly, *stride* is equal to 2^n and *n* changes from small to large value, where *n* belongs to positive integer and 2^n is less than the maximum iteration number of the *j*th loop level, then *indexj* changes from more deep loop level to more exterior loop level. Actually, the process of deleting voltage points is increasing the voltage scaling granularity, from more fine adjustment to more coarse, and balances the energy saving with voltage scaling overhead. At the

second step, we consider each inserting location as a voltage scaling point, and delete the voltage scaling point by setting *insert_or_not[i]* into zero. The detailed optimizing step is shown in Fig. 7. All the profile-guided time statistics are from *HEPTANE* tool.

First Step:	Second Step:
Input: the execution pattern of each loop in the most frequent execution case.	Input: the output from the first step.
Output: a configuration of voltage scaling points inserted into the loop.	Output: a configuration of voltage scaling points.
1 An initial optimal configuration without considering voltage scaling overhead.	1 Compute the energy consumption with *n* points by using the analytical model from [14]
2 Compute the energy consumption by using the analytical model from [14]	2 Compute the energy consumption with one point deleted (*n-1*).
3 Compute the energy consumption with *stride* incremented or *indexj* being more exterior loop level.	3 Compute the difference of the energy consumption between step 2 and step 1, and find out the minimum.
4 Compare the energy consumption for step 2 and step 3.	4 If the minimum is larger than zero, stop!
5 If step 2 has smaller energy consumption, stop!	5 Or else use the configuration with the minimum as the new configuration, update *n* (-1).
6 Or else repeat the steps from 2 to 6.	6 repeat the steps from 2 to 5.

Fig. 7. The improved optimizing method

5 The Experiment Environment and Results

We realize an experiment environment named *RTLPower* (Real-Time Low Power), which integrates static time estimation, cycle-accurate performance/power simulation, dynamic voltage scaling, and energy optimization. The front end is the modified *HEPTANE WCET* analysis tool [20], and the back end is the modified *Sim-Panalyzer* performance/power simulator [21]. The whole environment is based on the *StrongARM* architecture, as shown in Fig. 8. The gray regions are the modified or added modules. The front end of *RTLPower* receives the configuration information and C source code file with the annotation [20], the configuration information and source code file build the syntax tree of the application, and the code modification module modifies the source code by manipulating the syntax tree. The modified syntax tree is translated into context tree that is used to time estimation in *HEPTANE* tool. Using *HEPTANE* we estimate the worst case execution cycle of the application, the worst case execution cycle of all the loops, and the reduced worst case execution cycle of all the inserting points. The estimated time information is returned to the code modification module, and is used to create the complete syntax tree and output the DVS-enabled source code file. Integrated with the profile-guided optimization, we get the modified source code file and the final executable binary file.

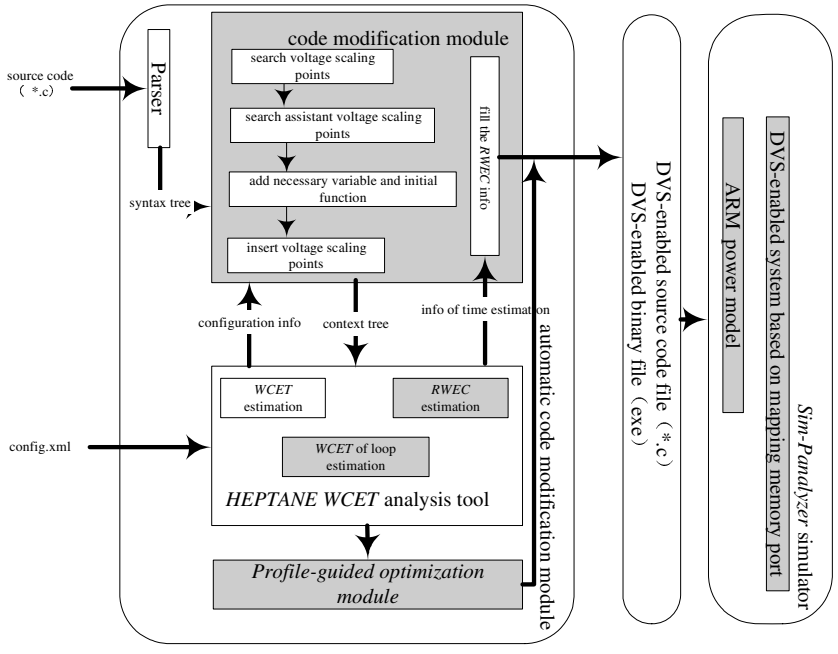

Fig. 8. *RTLPower* experimental envioment

Table 1. The performance parameters of *Sim-Panalyzer*

Fetch width	1	Decode width	1
Issue width	1	Commit width	1
RUU size	2	Lsq size	2
Int ALU	1	Int MUL	1
Flt ALU	1	Flt MUL	1
Mem Port	1	In-order issue	true
L1 data cache	16 sets, 32 bytes block, 32 ways, 1 cycle latency		
L1 inst cache	16 sets, 32 bytes block, 32 ways, 1 cycle latency		
TLB	32 sets, 4096 bytes page size, 32 ways, 30 cycles miss latency		

The back end of *RTLPower* cycle-accurately simulates the binary program and makes dynamic voltage scaling. It outputs time statistics, power statistics.

We use three typical applications of SNU-RT benchmark [22] from Real-Time Research Group, Seoul National University to analyze the realization and optimization of DVS. One of the applications is Adaptive Differential Pulse Code

Modulation (*adpcm*), and the whole application includes three stages: data initialization, data encoding, and data decoding. It includes 800 lines source code, many loop structures, condition structures and function calls, and the data input length is 2000. The second application is the fast fourier transform (*fft1k*), and the input data length is 1024. The final program is matrix multiplication (*matmul*). Its initial data size is 5x5, and we expand the size into 20x20. The performance parameters of *Sim-Panalyzer* are listed in Table 1. The voltage/frequency model comes from Intel Xscale [23], the frequency/voltage is listed in Table 2.

Table 2. The frequency and voltage of Intel Xscale

f(Mhz)	1000	800	600	400	150
V(V)	1.80	1.60	1.30	1.00	0.75

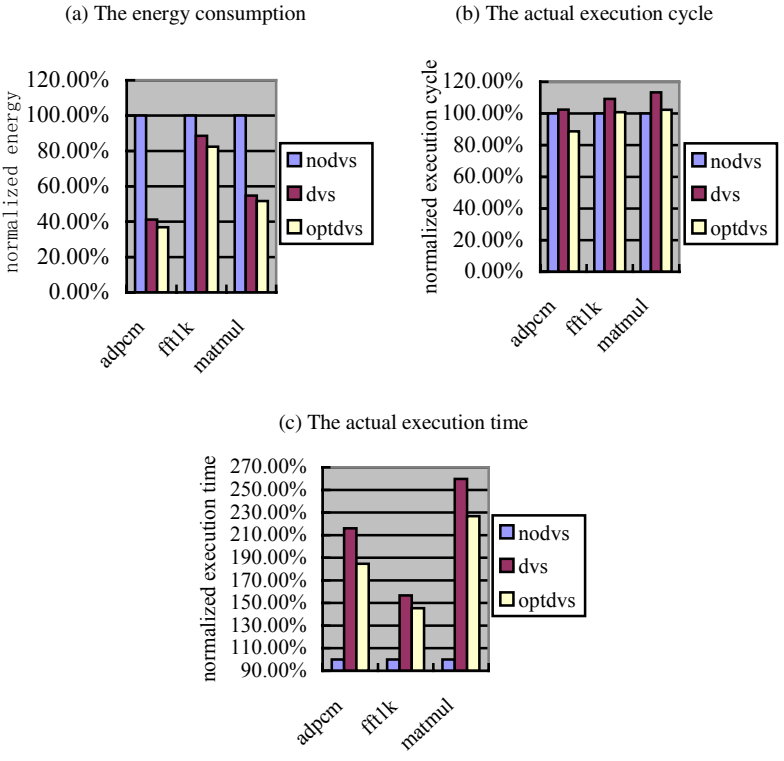

Fig. 9. The statistics of experimental results

We presents the experiment results in Fig. 9, where nodvs indicates the results of no voltage adjustment, dvs is the results with voltage adjustment, and optdvs is the results after profile-guided optimization. All the results are normalized to the maximum. The energy consumption without dvs and after voltage adjustment are

shown in Fig. 9(a), we can save 10%~60% energy consumption. After the profile-guided optimization, we further save 3%~6% energy consumption. In Fig. 9(b), we show the actual execution cycle, which indicates the incremented computation quantity. Generally speaking, dynamic voltage scaling leads to less computation quantity increment. For the application *adpcm*, after profile-guided optimization, fewer cycles are used, and we analyze that the result attributes to cache effect. Dynamic voltage scaling reduces energy consumption by slowing the execution and decreasing supply voltage, and in Fig. 9(c) we show the effect. The actual execution times are prolonged by 50~150%, and after optimization, both the execution cycle and time are reduced.

6 Conclusions

Based on the *WCET* (the worst case execution time) analysis tool *HEPTANE* and the performance/power simulator *Sim-Panalyzer*, we present a DVS-enabled simulation environment *RTLPower*, which integrates static WCET estimation, performance/ power simulation, automatically inserting the DVS code into a real application, and profile-guided energy optimization. By simulations of some real applications, we prove that the DVS technique and the profile-guided optimization technique significantly reduce energy consumption.

References

1. ITRS, “International Technology Roadmap for Semiconductors 2003 Edition,” Can get from http://public.itrs.net
2. Kanishka Lahiri, “Battery-Driven System Design: A New Frontier in Low Power Design, ” ASP-DAC/VLSI Design 2002, January 07 - 11, 2002, Bangalore, India.
3. T. Burd, T. Pering, A. Stratakos, and R. Brodersen, “A Dynamic Voltage Scaled Microprocess- or System,” in Proc. of IEEE International Solid-State Circuits Conference, 2000, pp. 294–295.
4. C.M. Krishna, Yann-Hang Lee, “Voltage-Clock-Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems,” IEEE TRANSACTIONS ON COMPUTERS, December 2003 (Vol. 52, No. 12).
5. Daniel Mosse, H. Aydin, B.R. Childers, R. Melhem, “Compiler-Assisted Dynamic Power-Aware Scheduling for Real-Time Applications,” Workshop on Compilers and Operating Systems for Low-Power (COLP'00), Philadelphia, PA, October 2000.
6. S. Lee and T. Sakurai, “Run-Time Voltage Hopping for Low-Power Real-Time Systems,” in Proc. of Design Automation Conference, 2000, pp. 806–809.
7. Dongkun Shin, Seongsoo Lee, Jihong Kim, “Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications,” In IEEE Design & Test of Computers, Mar. 2001.
8. H.Saputra, M. Kandemir, N.Vijaykrishnan, M.J.Irwin, J.S. Hu, C-H.Hsu, U.Kremer, “Energy-Conscious Compilation Based on Voltage Scaling,” In ACM SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems , June 2002.
9. Flavius Gruian, “Hard Real-Time Scheduling for Low-Energy Using Stochastic Data and DVS Processors,” In Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01 (Huntington Beach, CA, Aug. 2001).

10. Ana Azevedo, Ilya Issenin, Radu Cornea, "Profile-based Dynamic Voltage Scheduling Using Program Checkpoints," In Proceeding of Design, Automation and Test in Europe Conference (DATE), March 2002.
11. Nevine AbouGhazaleh, Daniel Mosse, B.R. Childers, R. Melhem, Matthew Craven, "Collaborative Operating System and Compiler Power Management for Real-Time Applications," in Proc. of The Real-time Technology and Application Symposium, RTAS, Toronto, Canada (May 2003).
12. Chung-Hsing Hsu, Ulrich Kremer, "The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction," in Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 38--48, June 2003.
13. Dongkun Shin and Jihong Kim, "Look-ahead Intra-Task Voltage Scheduling Using Data Flow Information," In Proc. ISOCC, pp. 148-151, Oct. 2004.
14. Huizhan Yi and Xuejun Yang, "Optimizing the Configuration of Dynamic Voltage Scaling Points in Real-Time Applications," In Proc. of PATMOS 2005, Sep 22-25,2005.
15. M. Fleischmann, "Crusoe Power Management: Reducing the Operating Power with LongRun," in Proc. of HotChips 12 Symposium, 2000.
16. Intel, Inc., "The Intel(R) XScale(TM) Microarchitecture Technical Summary," 2000.
17. AMD, Inc., "AMD PowerNow Technology," 2000.
18. Thomas D.Burd, Robert W.Brodersen, "Design Issue for Dynamic Voltage Scaling ," in Proc. of the 2000 international symposium on low power electronics and design, Rapallo, Italy, pages: 9-14.
19. Peter Puscher, Alan Burns, "A Review of Worst-Case Execution-Time Analysis (Editorial)," Kluwer Academic Pubilishers, September 24, 1999.
20. Antoine Colin, Isabelle Puaut, "Worst Case Execution Time Analysis for a Processor with Branch Prediction," Real-Time System, 2000, vol 18(2/3): 249-274.
21. Nam Sung Kim, Todd Austin, Trevor Mudge, "Challenges for Architectural Level Power Modeling," Book Chapter from Power Aware Computing, 2001.
22. SNU Real-Time Benchmarks. Get from http://archi.snu.ac.kr/realtime/benchmark/.
23. Dakai Zhu, Daniel Mosse and Rami Melhem, "Power Aware Scheduling for AND/OR Graphs in Real-Time Systems," IEEE Trans. On Parallel and Distributed Systems, vol. 15, no.9, pp.849-864, 2004.