Software Energy Optimization Through Fine-Grained Function-Level Voltage and Frequency Scaling

Carlo Brandolese  
Politecnico di Milano  
P.zza L. da Vinci, 32 – Milano (Italy)  
brandole@elet.polimi.it

William Fornaciari  
Politecnico di Milano  
P.zza L. da Vinci, 32 – Milano (Italy)  
fornacia@elet.polimi.it

ABSTRACT
This paper presents a methodology and a toolchain to perform estimation and optimization of the energy consumption associated to software execution on tiny embedded systems. The estimation phase is based on an ISA-level characterization of the target processor, while the optimization phase is made combining the estimation process with design space exploration in order to exploit fine-grained dynamic voltage and frequency scaling. The proposed approach operates at compile-time, with the granularity of single C functions and almost automatically augments the source code.

Categories and Subject Descriptors
C.4 [Computer Systems Organization]: Performance of systems; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems; C.0 [Computer Systems Organization]: Modeling of computer architecture; B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms
Design, Performance

Keywords
Power Optimization, Embedded Systems

1. INTRODUCTION
Traditionally, software consumption estimation is performed with energy- and cycle-accurate instruction-set simulators. These simulators are precise, but very computationally demanding and not useful to determine the consumption information of source-level entities. A first attempt to raise the abstraction level of the estimates [6] is made by coupling an ISS with coarse-grained estimates obtained with gprof, a profiling tool based on program counter sampling. This can resolve costs at the function level, but not at finer elements. The approach is good for obtaining rough estimates or comparing strongly different algorithms, but not helpful for optimization: experience shows that most of the time is spent inside loops entirely contained in one function, and the approach cannot resolve at that resolution. Splitting critical loops in functions just to allow measurement is not a solution because it seriously perturbs the estimates.

Another approach is the so-called compilation-based performance estimation. It exploits a real compiler (namely gcc) to obtain the control flow graph (CFG) of the code. The CFG is then annotated with information useful to derive a cycle-accurate model. SoftExplorer and similar tools implement a fast technique to estimate the consumption of data-dominated loops directly from the C source, based on functional-level modeling of the architecture and statistical parameters extracted from the assembly code. SIT is the first real 2-way source-level technique that redistributes estimates onto source-level entities, for the help of the developer. Here, like in our proposed approach, the source code is conceptually instrumented, though in a different way. More precisely, the input C code is promoted to C++ and overloaded instrumented versions of all the operators are provided. The estimation approach proposed in this this paper is an evolution of a methodology and its related toolchain already presented by the authors in a previous work [4]. Two are the major novelties of the models and of the toolchain described in this paper:

- The new methodology expresses the energy costs of the basic entities of the intermediate representation in term of effective capacitance, rather than as average current absorptions per clock-cycle. This choice allows accumulating energy figures independently from the actual clock frequency and core supply voltage. This approach is crucial for exploring different voltage/frequency operating modes, since it decouples the process of capacitance and timing characterization of individual functions from the actual computation of overall energy and time figures.

- The toolchain totally decouples the the estimation of the effective capacitance and the number of clock cycles for the execution of individual functions from the overall voltage and frequency dependent time and energy figures. In this way capacitance and number of clock cycles associated to the execution of a function operating on fixed data can be calculated once and for all in a first phase, while leaving to a second post-processing step the actualization of such figures for function-specific frequencies and voltages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CODES+ISSS’12, October 7-12, 2012, Tampere, Finland.  
Copyright 2012 ACM 978-1-4503-1426-8/12/09 ...$15.00.
The core part of this paper concerns energy optimization, and in particular a methodology to exploit voltage and frequency scaling capabilities of the target core. The proposed approach is targeted to small, constrained embedded application built in absence of an operating system and combines the estimation methodology described so far—implemented by the SWAT toolchain—with an optimization tool for design space exploration—the MOST optimization engine—both developed at Politecnico di Milano.

The key idea is to explore possible assignments of operating modes to individual functions, find the optimal assignment, and (almost) automatically augment the source code of the functions in such a way that mode switching is performed by the function themselves.

2. MODELS

2.1 Estimation model

The basic ideas supporting the proposed approach have been explored by the authors in previous works [3] and finalized in [4]. The new approach described in this paper uses an intermediate representation based on the assembly-level LLVM code produced by the open-source project LLVM Compiler Infrastructure [2]. The modeling approach described in the following mimics the methodology proposed in [4], but uses the concept of effective capacitance to express the basic energy figures on which the estimation is founded. The estimation is decomposed in three subsequent phases:

1. Source code modeling. This first modeling phase decouples the influence of the static structure of the source code from the specific target architecture from its dynamically varying operating conditions (voltage and frequency) and from the input runtime data.

2. Dynamic behavior modeling. The dynamic behavior is introduced in the model through basic-block profiling at intermediate-representation level (i.e., LLVM code). Such a profiling can be accomplished by executing the code on a host machine, possibly resorting to some support libraries that model hardware devices.

3. Target machine modeling. Finally, the actual timing and energy characteristics of the target architecture are combined with static and dynamic models to derive the final estimate. To this purpose it is necessary to derive a target-specific cost model of the LLVM instruction set and characterize the voltage and frequency transitions supported by the core under consideration.

2.1.1 Source code and dynamic models

Given a program run, that is the combination of a program and a specific set of input stimuli, let $e^{(c)}$ and $e^{(f)}$ be consumed energy and the time necessary for its execution. The LLVM intermediate representation can be seen as a list of basic-blocks, each constituted by LLVM instructions of different types. Let $B$ be number of basic blocks and $L$ be the number of instructions composing the pseudo-assembly LLVM instruction set. The representation matrix $R = [R_{i,j}]$ describes the combination of instructions belonging to each basic block. More precisely, $R_{i,j} = n$ means that the $i$-th basic block contains $n$ LLVM instructions of type $j$. The matrix $R$ is thus a static representation of the source code. Note that, for the purpose of this model, the ordering of instructions within a basic block is not relevant. More accurate models proposed in literature also account for inter-instruction effects, and thus require to know the exact ordering of instructions in the program. In order to be as independent as possible from the actual target processor and instruction set, the approach described here operates directly on the LLVM intermediate code, which will be translate into the target code in a subsequent phase. This prevents obtaining information about the actual ordering of target instructions.

Inter-instruction effects at the LLVM level are thus irrelevant since show extremely limited correlation—if any—with the effect associated to the final translation. The instrumentation process is performed by adding suitable tracing function calls to each basic-block. The instrumented program is then compiled, linked and run on the host machine. As a result, a profile $p = [p_1, p_2, \ldots, p_K]$ is generated. In such a profile $p_i$ indicates the execution count of the $i$-th basic-block. Combining the profile $p$ and the representation matrix $R$, the vector of execution counts of all instruction types is obtained. Mathematically:

$$s = p \cdot R = [s_1, s_2, \ldots, s_L]$$

where $s_i$ is the overall execution count of the instruction of type $j$ in the specific program run under consideration. Since each instruction of the intermediate language corresponds to one ore more assembly instructions executed by the target architecture, the evaluation of the overall cost of a program run requires decomposing the intermediate instructions into such elementary contributions. The elementary operations considered at this stage depend on the target architecture model and more precisely on the target assembly language. Let $K$ be the number of instructions of the target assembly language and $T = [T_{i,j}]$ be the translation matrix with $L$ rows corresponding to the types of LLVM instructions, and $K$ columns corresponding to the different target assembly instructions. The element $T_{i,j} = n$ models the fact that the LLVM instruction of type $i$ requires $n$ instruction of type $j$ of the target assembly. Of course, the exact form of the target code resulting from the translation of an LLVM instruction cannot be known in advance. This is due to two main reasons. First of all, the LLVM code is in static single assignment (SSA) form, and thus uses an unlimited number of virtual registers, while the actual code will be limited by the number of register available on the target architecture. This limitation of the available registers will result in memory spilling, that is, additional load/store instructions. Secondly, even though the first pass of the LLVM-to-target code translation is deterministic and template-based, the subsequent optimization pass will strongly modify the code. For these reasons, the translation matrix has a statistical meaning only. This matrix is the result of the characterization phase, described in in Section 2.1.2 and can be also viewed as a collection of $L$ row vectors $T_i$. Each vector represents the statistical translation model of a specific LLVM instruction. Indicating with $L_m$ the LLVM instruction of type $m$ and with $T_i$ the target assembly instruction of type $i$, the cost of $L_m$ for the specific target architecture is:

$$\text{cost}(L_m) = \sum_{i=1}^{K} T_{m,i} \cdot \text{cost}(T_i)$$

The target processor is supposed to be characterized in terms of execution time and energy consumption with the granu-
larity of single assembly instruction. It is worth noting that, while the actual average execution time of each instruction in generally known and publicly available, the energy consumption is often not known or not available for single instructions. In this case, accepting a loss of accuracy, average energy consumption figures can also be used. The proposed model, in fact, only accounts for such figures in the final phase. This decouples the formal static and dynamic modeling phases from the final physical characterization.

Each target instruction $T_i$ is thus characterized by its average execution time $n_i$, expressed in clock cycles, and its effective capacitance $c_i$, expressed in farad. Furthermore, the instantaneous operating conditions of the microprocessor—referred to as mode in the following—is characterized by the power supply voltage and the clock frequency. Indicating with $M_i$ a mode of the core, let $V_i$ and $f_j$ be the corresponding voltage and frequency respectively. The execution time and energy of instruction $T_i$ in mode $M_j$ are thus:

$$K^{(t)}_{ij} = \frac{n_i}{f_j} \quad K^{(p)}_{ij} = \frac{1}{2} c_i V_i^2 f_j$$

Combining these expressions, finally, gives the energy consumption function $T_i$ in mode $M_j$:

$$K^{(c)}_{ij} = K^{(p)}_{ij} K^{(t)}_{ij} = \frac{1}{2} c_i V_i^2 n_i$$

Timing and energy figures for each target instruction $i$ and each processor mode $j$ are thus collected into two matrices $K^{(t)}$ and $K^{(c)}$. The current operating mode $m$ of the processor is represented by means of the mode selection vector $M_m = [\mu_{m,j}]$, where $\mu_{m,j} = 1$ if $m = j$ and 0 otherwise. Combining all equations derived so far gives the total execution time and energy consumption of the task:

$$e_m^{(c)} = p \cdot R \cdot T \cdot K^{(c)} \cdot \text{diag}(M_m)$$

$$e_m^{(t)} = p \cdot R \cdot T \cdot K^{(t)} \cdot \text{diag}(M_m)$$

These equations summarize our estimation methodology.

### 2.1.2 Target Architecture Characterization

The model described in the previous section relies on several matrices that need to be determined by means of a suitable procedure, in particular:

- $R$. The representation matrix is determined by the front-end portion of the toolchain described in Section 4. The source code is compiled into LLVM code, which is in turn transformed into a model file describing the mix of LLVM instructions per each basic block.

- $T$. The translation matrix is constructed by juxtaposition of the translation model vectors $T_i$ for each LLVM instruction. Such vectors express the static correlation between the mix of instructions used by the LLVM compiler and the actual mix of assembly instructions generated by the target compiler. The procedure adopted to determine the individual translation model vectors is the core of this section.

- $n_i$, $c_i$. These figures are provided by the target architecture manufacturer. They express, for each instruction of the target assembly, the average execution time in clock cycles and the effective capacitance.

- $p$. The profile vector models the dynamic behavior of an application run, i.e. a program and a specific set of input data. It is derived by the back-end portion of the toolchain, as described in Section 4.

In this section, we shortly describe the methodology used to determine the translation vectors for each LLVM instruction $L_m$. This procedure has been described in detail in [4].

The key idea of the methodology consists in computing the correlation between the counts of each LLVM instruction of the LLVM-compiled code and the counts of the target instructions of the same code compiled with the target compiler. This requires first identifying a suitable, large enough, set $S = \{S_0, S_1, \ldots, S_N\}$ of simple but varied source codes to be used for learning. Let then $L_{i,j}$ be the number of LLVM instructions of type $j$ in the LLVM code corresponding to the source code $S_i$ and $D_{i,k}$ be the number of target instructions of type $k$ in the destination assembly code corresponding to the same source code $S_i$. The analysis of all source codes for each instruction leads to the over-constrained system:

$$L_{i,j} = \sum_{k=1}^{K} D_{i,k} \cdot x_{j,k}$$

in the $N \times K$ unknowns $x_{j,k}$. This set of equations can be rewritten into matrix form as $L_i = D x_j$, where $L_i$ and $x_j$ are vectors representing the LLVM instruction count and the coefficient of the desired model, respectively, and $D$ is the matrix of target instruction counts over the entire set of source codes. Since it is not realistic to assume that all target instructions will participate in the translation of each single LLVM instruction, we restrict the set of target instructions that can contribute to the model of each specific LLVM instruction by defining an initial model vector $M_j = \{m_{j,1}, m_{j,2}, \ldots, m_{j,K}\}$ of coefficients $m_{j,p} \in \{0,1\}$. A value $m_{j,p} = 1$ indicates that the target instruction of type $p$ participates to the model of the LLVM instruction of type $j$. The general equation $L_i = D x_j$ can be modified to include the initial model vector by simply transforming the vector $M_j$ into a diagonal matrix and multiplying it by the coefficient matrix $D$, that is:

$$L_i = D \text{diag}(M_j) x_j = D_j x_j$$

A further restriction necessary to preserve the physical meaning of the mode is that the coefficients $x_j$ must be positive. This requirement implies solving the problem of Equation 8 in non-negative least-square sense, that is:

$$\|D_j x_j - L_i\|^2 \quad \text{with} \quad x_j \geq 0$$

To this purpose, we adopted the method described in [1], Ch. 23, p. 161. Once the system has been solved, the translation matrix can be built as $T = [x_0^T x_1^T \cdots x_L^T]$.

### 2.2 Operating modes model

The target processors, as anticipated in Section 2.1, provides a set of operating modes, which will be referred to in the following as explicit modes. Each function can be assigned a specific explicit mode and, as a consequence, the toolchain will augment the $C$ source code by inserting platform-specific library function calls devoted to switch to the selected mode on entering the function and back to the previous mode, on exiting. In addition to the explicit modes, we define the two implicit modes referred to as force and inherit, as described in the following.
**Figure 1:** Mode assignment effects

**Force.** When the mode of a function is set to force to a specific explicit mode, all its callees will be executed in the same operating condition as the caller, regardless of their specific explicit assignments.

**Inherit.** The inherit mode has, in a sense, a dual meaning. It specifies that the mode of a function is not explicitly set, but is rather inherited from its caller.

These special modes provide a simple yet powerful means to have a function being executed in different modes, depending on its context, thus providing more flexibility to the approach. Let us consider, as an example, a task constituted by three functions $f_1()$, $f_2()$ and $f_3()$ and target processor with two modes only: 1 and 2. The timing diagram of Figure 1 shows the effects of three different mode assignments on the processor operating conditions. In the figure, the labels $X_1$ and $X_2$ indicate explicit mode assignments, $F_1$ the force mode, and $I$ the inherit mode. Furthermore, the solid line, indicates that a function is executed in mode 2 while the dotted line represents execution in mode 1.

A part from the obvious effects of explicit assignments shown in Figure 1-(a), it is interesting to observe the behavior resulting from forcing and inheriting modes. To this purpose we concentrate the attention on function $f_3()$. When $f_2()$ forces the mode 1 and calls $f_3()$, Figure 1-(b) shows that the explicit mode assignment of $f_3()$ is ignored. On the other hand, when called directly from $f_1()$, $f_3()$ is executed in mode 2, as specified by its explicit assignment. Furthermore, observing the diagram in Figure 1-(c), it can be noted that the operating mode in which $f_3()$ is executed is always that of its caller. Though the behavior in these two last cases is the same, it is obtained in two dual ways: in the first case the caller imposes its mode to all callees, while in the second is the callee that delegated the decision on the operating mode to its caller.

### 3. Optimization Model

The goal of the optimization is to find the assignment of a mode to each function in such a way to minimize the overall energy of a program run with a constraint on the maximum allowed time for the task. Let us now consider a call of function $F_j$ from function $F_i$. Given a specific mode assignment $M$, function $F_i$ will be executed in mode $m_i$ and function $F_j$ in mode $m_j$. In the following we will indicate such a call with the notation:

$$[i, m_i] \rightarrow [j, m_j]$$

(10)

Section 2.1 provides the background for the estimation of an entire program run, but the same model, methodology and toolchain can be used to obtain cost estimates for individual functions. Let thus denote with $c_{k,m}^{(e)}$ and $c_{k,m}^{(t)}$ the energy and execution time of function $F_k$ with the processor in operating mode $m$. Furthermore, let denote with $\tau_{m_i,m_j}$ the energy and time associated with the transition from mode $m_i$ to mode $m_j$. The total execution time and energy consumed for the execution of the call $[i, m_i] \rightarrow [j, m_j]$ with the mode assignment $M$ can thus be expressed as:

$$T([i, m_i] \rightarrow [j, m_j], M) = c_{j,m_j}^{(t)} + \tau_{m_i,m_j}^{(e)} + \tau_{m_i,m_j}^{(t)}$$

(11)

$$E([i, m_i] \rightarrow [j, m_j], M) = c_{j,m_j}^{(e)} + \tau_{m_i,m_j}^{(e)} + \tau_{m_i,m_j}^{(t)}$$

(12)

Adding such total costs for all the calls executed by the task $T$ gives the total execution cost of the task run, that is:

$$T(M) = \sum_{[i, m_i] \rightarrow [j, m_j] \in T} T([i, m_i] \rightarrow [j, m_j])$$

(13)

$$E(M) = \sum_{[i, m_i] \rightarrow [j, m_j] \in T} E([i, m_i] \rightarrow [j, m_j])$$

(14)

The optimization goal can now be formally stated as finding an assign $M_{opt}$ such that:

$$E(M_{opt}) = \min_{M \in M^*} E(M) \land T(M) < T_{max}$$

(15)

where $M^*$ is the set of all possible assignments and $T_{max}$ is the maximum execution time allowed for the task.

Considering $N_M$ possible processor operating modes and $N_F$ functions, the exact solution of the problem requires examining $N_M^N_F$ assignments. Given the exponential complexity, this problem becomes soon intractable. For 4 modes and 20 functions, for example, the number of assignments is close to 3.5 billions, which makes using a heuristic design space exploration approach necessary.

### 4. Tool Flow

This section describes the estimation and optimization flows. The former is detailed in [4], while the exploration methodology is described in [5].

#### 4.1 Estimation

The implemented estimation flow is based on the LLVM compiler infrastructure, upon which a completely custom toolset called SWAT (SoftWare Analysis Toolset) has been developed. A simplified view of the portion of the flow strictly related to the estimation process necessary for the problem being studied in this paper is outlined in Fig. 2. The input is the set of C source files ($\bullet, .c$) collecting the code of task being considered, a model of the target CPU (cpu.lib) and an assignment of modes to functions (task.modes). Performing a sequence of transformations the flow produces the energy and time estimates $T$ and $E$. Note that the mode assignment and the target processor model files are fixed.

The transformation performed by the tools of the flow have been collected into four phases, indicated by numbered black boxes, and are detailed in the following.
1. **Front-End.** This phase compiles each source file into architecture-independent LLVM assembly code, which is then used to build a model (*.bbmodel) of each basic block consisting of the list of op-codes, functions called, size, execution time in clock cycle and effective capacitance. Data for timing and energy characterization is in the target CPU library (cpu.lib), which is the result of characterization (Section 2.1.2).

2. **Instrumentation.** Instrumentation is performed by first enriching each basic block of the LLVM code with all the relevant figures in the form of a special comment (meta-instrumentation), then by translating the comments into actual calls to tracing functions based on expansion rules collected into an instrumentation library. The output of this phase is a new, instrumented, LLVM assembly file (*.i.ll).

3. **Back-End.** The back-end of the SWAT flow performs two main operations. First, it translates all the instrumented LLVM files into host assembly code, which is then assembled and linked into an executable program. Secondly, it runs the executable and collects the execution trace (bbtrace) consisting of a list of the identifiers of the basic blocks that have been executed.

4. **Post-Processing.** The post-processing phase analyzes the execution trace and combines the dynamic information with the static costs models, accounting for the specific operating modes specified in the allocation file (task.modes). This produces the total timing and energy of the specific run of the task.

This estimation flow is then combined with the optimization engine MOST that performs design space exploration over the possible mode assignment. It is worth noting that steps 1-3 of the estimation flow need not to be repeated for each and every assignment. They are, in fact, performed only once with the goal of producing an execution trace and a set of cost models. Steps 1-3 (and in particular step 3, that involves task execution) are much more time-consuming than the post-processing phase only. The proposed toolchain is thus efficient enough to enable design space exploration with simulation-in-the-loop.

### 4.2 Optimization

The optimization flow, sketched in Fig. 3, is built around the design exploration engine MOST.

5. **Design Space Exploration.** This tool requires a configuration file (task.dse) specifying which are the parameters and which values each parameter can assume. In our case the parameter are the modes \( m_i \) of each function and the values are integers in the range \([1; N_M]\) corresponding to the target processor modes. Based on this configuration, the DSE engine generates a specific mode assignment (task.modes) which is fed as input, together with the execution trace and the basic block models, to the SWAT post-processor. The execution time and energy estimated by SWAT are used by MOST to selects a new, potentially better, assignment. This loop is repeated until a satisfactory assignment (task.opt.modes) is found.

6. **Code augmentation.** Using a set of predefined macros and a simple code generator, this tool adds to the original source code, at the beginning and at the end of each function, the suitable code performing mode switching. The augmentation mechanism is described in further detail in Section 4.3.

The two toolchains depicted in Fig. 2 and 3 have been used to produce the experimental results described in Section 5.

### 4.3 Code generation

The code generation is the last step of the optimization flow and its goal is to augment the original source code with calls to suitable—and user-definable—APIs devoted to changing the operating mode of the processor on entry and/or on exit of a function.
This process is currently semi-automated, as it requires adding two macros at the beginning and at the exit of each function that is considered in the exploration process. Considering, for example, a function:

```c
int foo( int x, int y ) {
    // --------------------------
    // Declarations
    // --------------------------
    // Function body
    return some_var;
}
```

and assuming that the function as a single exit point, the only task left to the programmer is to modify the function definition as follows:

```c
int foo( int x, int y ) {
    // --------------------------
    // Declarations
    // --------------------------
    VFS_ENTER(foo)
    // --------------------------
    // Function body
    // --------------------------
    VFS_EXIT(foo)
    return some_var;
}
```

The expansion of the two macros generates new code that depends in turn on other macros built based on the function name passed as argument, which, in our example, would be `VFS_MODE_foo`. Since functions can not only be assigned explicit modes, but also can be defined as forcing or inheriting the operating mode of caller/callee, it is necessary to implement a sort of mode stack where to save the mode of the current function before entering one of its callees and restore this mode on exit. Rather than using a separate stack, our mechanism is based on four entities, namely:

- A global variable `vfs_cm` storing the current mode. The variable is static in a support library that need to be compiled along with the application.
- A global variable `vfs_fm` indicating whether the current mode is being forced or if it is explicit/inherited. This variable is also static in the support library.
- A variable `vfs_sm`, local to each function, holding the saved mode, i.e. the current mode upon function entry.
- A macro `VFS_SET_MODE(m)` that is platform specific and will be expanded to a call to the suitable function exposed by the target API and responsible of changing the operating mode. Such a function will usually write the relevant CPU registers.

Thanks to the local variable added to each function, a separate stack for modes is not necessary as it is distributed into the activation frames of the function itself. Using these variables and exploiting the macros, the code of the original function is transformed into that shown in Figure 4.

![Figure 4: Example of augmented code](image)

The experimental results presented here refer to a specific target processor, namely the STMicroelectronics ultra low power core ReISC 4. Since this core is still in development, timing and energy characterization is not yet publicly available. It has been disclosed to Politecnico di Milano under the agreement of the COMPLEX Project. For this reason, energy and timing figures appearing in the graphs of this section have been scaled by a constant factor in order to not disclose proprietary information. The ReISC 4 core provides dynamic voltage and frequency scaling capabilities over three different modes.

In order to validate the approach on a large number of differently structured tasks, synthetic code has been used. To this purpose a parametric tool for code generation has been developed. It can generate random programs based on the parameters summarized in Table 1 along with the ranges used to generate the specific tasks for which results are reported.

Let us start considering a simple example, with three functions only. In this case the possible assignments are $3^5 = 125$. Since the exploration engine do not perform an exhaustive analysis, much fewer assignments have been generated, as shown in the plot of Figure 5. As it can be noted, to a reduction of the execution time corresponds an increase in the energy consumption. In this example the execution...
time constraint was set to 1.65 µs. The solution found, highlighted in the plot was characterized by an execution time of 1.626 µs and an energy consumption of 412 nJ. This correspond to an average power consumption of 253 µW, as the plot of Figure 8.

The results of exploration applied to a more complex task are reported in Figures 6 and 9. The task analyzed is composed of 5 functions for an overall dynamic call count of 9782. In this case the time constraint was set to 11.5 ms and the optimal solution found has an energy consumption of 2.94 µJ and corresponds to an execution time of 11.15 ms.

The last results reported in detail refers to a much more complex task with 20 functions and an overall dynamic call count slightly greater than 1.49 million. It must be noted that in this case the portion of the design space of all possible assignments has been explored in a minimal part, that is in 3800 points out of the possible $9.53 \times 10^{13}$. As Figures 7 and 10 show for execution time and energy consumption, the overhead introduced by the operating mode transitions, though much smaller than the figures associated to the actual execution of the task, cannot be neglected. In the worst cases, in fact, the timing overhead impacts on the overall execution time for 27.9%, while for the energy the impact in worst case is limited to 8.1%.

Finally, the results obtained applying the proposed optimization methodology to a set of 33 randomly generated tasks are reported in Figure 11. The plot shows the energy consumption of the optimized task (black bars) with that obtained maintaining the system either in the highest voltage/frequency mode (white bars) or in its deepest low-power modes. It must be noted that the optimized tasks—and of course the tasks run in full active mode—do respect their deadlines, while the tasks run in the lowest power mode do not. The energy gains obtained by the mode allocation tech-
6. CONCLUSIONS

This paper extended and adapted a previous estimation methodology to allow the analysis of execution time and energy consumption of software for processors supporting dynamic voltage and frequency scaling. The methodology has been implemented in the SWAT toolchain and integrated with the design exploration engine MOST to minimize the execution energy of a task under timing constraints by finding a sub-optimal assignment of operating modes to individual functions. The experimental results obtained on a large number automatically generated tasks show that an average 20% energy saving can be obtained, with only a trivial manual intervention on the source code.

Acknowledgements

This work has been partially supported by the EU integrated project COMPLEX under grant FP7-247999.

7. REFERENCES