Design Documentation for EMSIM

T. K. Tan†, A. Raghunathan‡, and N. K. Jha†

† Dept. of Electrical Eng., Princeton University, NJ 08544. Email: {tktan,jha}@ee.princeton.edu

‡ NEC, C&C Research Labs, Princeton, NJ 08540. Email: anand@ccrl.nj.nec.com

I. Design of the Simulator

To enable an OS such as Linux to run in the simulator, we need to model the various system modules in sufficient detail. Besides modeling the core components such as instruction decoder, pipeline, caches and memory management unit (MMU), other hardware components essential for the functionality of an OS should also be modeled correctly. For example, we need to have timers to set the pace for task scheduling. We need to model *universal asynchronous receiver transmitters* (UARTs), since Linux can use one of the serial ports as the console output. Most importantly, we also need to have an interrupt controller to service the interrupt requests from the timers, UARTs and the internal software interrupts.

In the following sub-sections, we discuss various aspects of the design in detail.

A. Basic Features

The hardware structure of the system that we model in our simulator is shown in Fig. 1. The corresponding simulation framework is shown in Fig. 2. Shown on the right half of Fig. 2 are the simulation models for all the sub-systems featured in Fig. 1, whereas the sequence of steps involved in using the simulation framework is shown on the left. As shown in Fig. 2, the simulator includes the following components:

1. A simulator for the StrongARM core, consisting of an *instruction-set simulator* (ISS), simulation models for 16K D-cache and I-cache, and a memory management unit (MMU).
2. Simulation model for an interrupt controller.
3. Simulation models for two timers.
4. Simulation models for two UARTs conforming to the 8250 series. By default, we direct the transmitter of UART0 to the host terminal. We also connect the transmitter/receiver of UART1 to a TCP socket so that two instances of the simulators can communicate with one another through their UART1's, respectively. This feature allows simulations that involve two machines communicating with one another

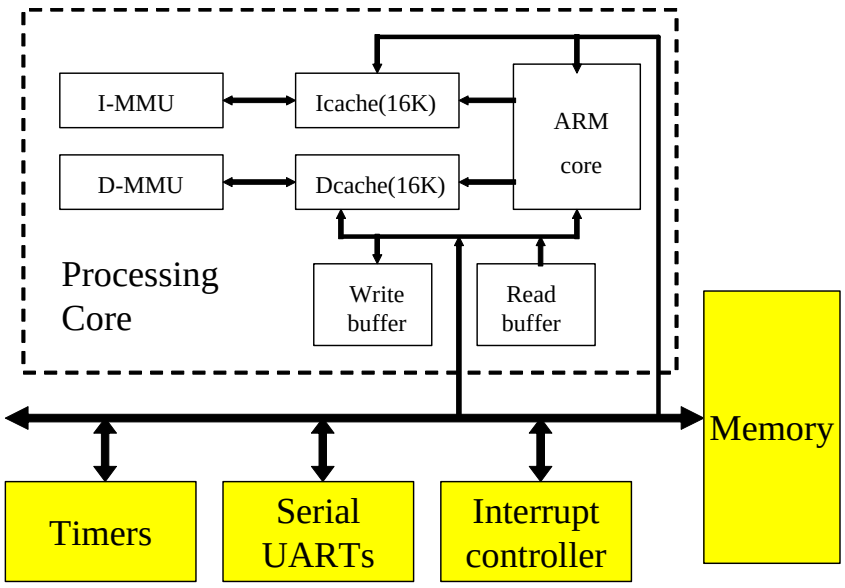

Fig. 1. Modeled embedded system

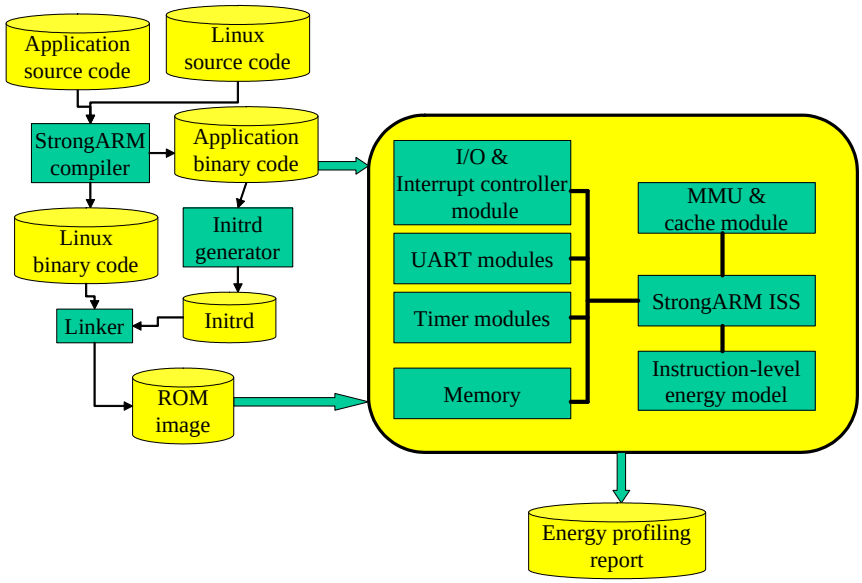

Fig. 2. Energy analysis framework

through their UART's.

5. Simulation model for 32 Mbytes of memory.

We have modeled each component in sufficient detail to enable a version of Linux OS to execute on it. The Linux OS that we adapted for the purpose of simulation has the following features:

1. It is *arm-linux* version 2.2.2, configured for the EBSA-110 platform.
2. We have configured the *arm-linux* to mount an initial ramdisk (*initrd*) as the root filesystem. The user application code is pre-installed into the *initrd*. At the end of the kernel initialization steps, the user application program is loaded from the ramdisk.
3. We have also configured the *arm-linux* to use UART0 as the console. Output from the kernel as well as from the user application can thus be displayed on the host terminal since the transmitter of UART0 is redirected to the host terminal.

B. Usage of the Simulator

The flow diagram depicting the usage of the simulator is shown in the left part of Fig. 2. The boxes with dark shading represent the auxiliary tools employed in the flow, whereas the cylinders with light shading represent the objects that are being manipulated at various stages. Given the *arm-linux* source code, we generate the *arm-linux* object code using a cross-compiler. In parallel, we also generate the user application object code from the application source code using the cross-compiler. As mentioned in the previous sub-section, we need to pre-install the user application code into the *initrd*. To do that, we use an *initrd* generator. The output of the *initrd* generator is a compressed image of the *initrd*. After that, a linker is used to link the *arm-linux* object code and the compressed *initrd* image into a final ROM image.

At the start, the simulator loads the ROM image into the memory. Note that the application object code is also loaded by the simulator, as depicted in the flow diagram. This is necessary even though the application is already pre-installed in the *initrd* because we need the function symbol information directly from the application code to enable process and function-level energy profiling.

C. Energy Modeling in the Simulator

We now describe the energy models used in the various components of the simulator. Most of the concepts presented in this sub-section are adapted from previous work on software and system energy estimation. The description is included for the sake of completeness. Since the main purpose of building the simulator is to analyze the energy consumption from the software perspective, we do not resort to analytical energy models as described in [1]. In fact, Simunic *et al.* [2] have demonstrated that reasonably accurate (within

5% error) power estimation can be obtained even if the power models of the constituent components are inferred directly from the data-sheet information for the system components. In our work, we follow this philosophy for most components of the simulator except for the processor, for which we have obtained the StrongARM instruction-level energy models from [3]. The accumulated energy consumption of the embedded system for the duration of a program execution is calculated as follows:

$$E_{sys}^{T} = E_{proc}^{T} + E_{idle}^{T} + E_{mem}^{T} + E_{uart}^{T} + E_{peri}^{T} \quad (1)$$

E_{sys}^{T} = total accumulated system energy consumption
E_{proc}^{T} = accumulated processor active energy consumption
E_{idle}^{T} = accumulated processor idle energy consumption
E_{mem}^{T} = accumulated memory energy consumption
E_{uart}^{T} = accumulated UART energy consumption
E_{peri}^{T} = accumulated energy consumption of other peripherals

Each of these contributions is explained in the following sub-sections.

C.1 Accumulated processor energy consumption (E_{proc}^{T} and E_{idle}^{T})

The accumulated processor energy consumption E_{proc}^{T} is calculated based on the instruction-level methodology for power/energy modeling:

$$E_{proc}^{T} = \sum_{i=1}^{N_{instr}^{T}} E_{proc}[instr\_type(i)] * N_{cyc}(i) \quad (2)$$

N_{instr}^{T} = total number of instructions executed
$instr\_type(i)$ = instruction type (index) of the i^{th} executed instruction
$E_{proc}[instr\_type(i)]$ = processor energy consumption per cycle when executing instruction type $instr\_type(i)$
$N_{cyc}(i)$ = number of cycles required to execute instruction i

The instruction-level energy models from the MIT μAMPS project on SA1100 instruction current profiling experiment [3] already include the energy contributions of all parts of the processor core, including cache, MMU, clock generator, timer, interrupt controller, *etc.* Following the original methodology described in [4], the instruction-level energy models are applicable when cache hits occur. The data provided in [3] are in the form of instruction current consumption. To calculate the corresponding energy consumption per processor core cycle when a particular instruction executes, we use the following conversion:

$$E_{proc} = \frac{V_{dd} * I_{instr}}{f_{clk}} \quad (3)$$

In the experiment, the core supply voltage V_{dd} used was $1.46V$, and the clock frequency f_{clk} was $206MHz$. The instruction current I_{instr} ranges from $0.165A$ to $0.237A$. Using Equation (3), we determine that the energy consumption of the processor core in each cycle ranges from 1.17 nJ to 1.68 nJ, depending on the instruction being executed. The detailed E_{proc} data categorized by instruction groups are listed in Table I.

Our simulator also supports StrongARM's *idle mode*. In the *idle mode*, the clock to the processor core is stopped, but all other on-chip resources, such as clock generator, timers, interrupt controller, UARTs, power manager, *etc.*, are still active. When an interrupt occurs, the core is re-activated.

When the processor is running the Linux kernel without a workload, the idle task in the kernel kicks the processor into *idle mode*. From [5], we obtain the idle mode power consumption of the processor chip to be 65 mW for a clock frequency of 206 MHz. Since the clock generator is still running, we can calculate the total idle energy based on the clock tick:

$$E_{idle}^{T} = P_{idle} * T_{cyc} * N_{idle\_cyc} \quad (4)$$

TABLE I
E_{proc} DATA CATEGORIZED BY INSTRUCTION GROUPS

Instruction group	E_{proc} (nJ)
ADC	1.23
ADD	1.26
AND	1.23
BIC	1.23
CMN	1.28
CMP	1.28
EOR	1.27
MOV	1.25
MVN	1.25
ORR	1.27
RSB	1.28
RSC	1.28
SBC	1.28
SUB	1.27
TEQ	1.27
TST	1.46
MLA	1.46
MUL	1.31
LDR	1.47
LDRB	1.47
LDRBT	1.31
LDRT	1.31
STR	1.63
STRB	1.62
STRBT	1.62
STRT	1.63
LDM	1.68
STM	1.62
MRS	1.23
MSR	1.20
SWP	1.17
B	1.20
NOP	1.22

where idle power $P_{idle} = 65\ mW$, T_{cyc} is the core clock period, and N_{idle_cyc} is the total number of core clock cycles during which the processor stays idle.

When cache misses occur, additional energy is consumed as a result of external bus and memory access. We address this next.

C.2 Accumulated memory energy consumption E^T_{mem}

We assume that our memory part is the same as that used in the Compaq Western Research Lab's Itsy pocket computer v1.5 [5, 6]. The energy consumption of the memory part in each bus clock cycle can be extracted from the data given in [5]. Note that clock switching [7] is normally enabled by the *arm-linux* kernel, hence the bus clock frequency is half of the core clock frequency. From [5], we estimate the memory

energy consumption for each bus clock cycle to be 4.70 nJ (denoted as E_{mem}). Given that, the accumulated memory energy consumption is calculated as:

$$E^T_{mem} = E_{mem} * N_{mem\_cyc} \tag{5}$$

where N_{mem_cyc} is the number of memory cycles in which the memory part is active.

C.3 Accumulated UART energy consumption E^T_{uart}

According to [5], the additional power consumption incurred when UARTs are enabled is fairly constant at approximately 44 mW. At a core clock frequency of 206 MHz, we estimate the energy consumption of the UART module in each core cycle to be 0.21 nJ (denoted as E_{uart}). Given E_{uart}, we calculate E^T_{uart} as:

$$E^T_{uart} = E_{uart} * N_{uart\_cyc} \tag{6}$$

where $N^T_{uart\_cyc}$ is the total number of core cycles in which the UART is active. Note that the UART can be active independent of whether the processor core is in the idle mode.

C.4 Accumulated peripheral energy consumption E^T_{peri}

Energy models for other peripherals can be added in the future in the same manner as for the UART. That is, the accumulated peripheral energy consumption E^T_{peri} can be calculated as

$$E^T_{peri} = E_{peri} * N_{peri\_cyc} \tag{7}$$

where E_{peri} is the per cycle energy consumption of the peripheral, and N_{peri_cyc} is the total number of core cycles in which the peripheral is active.

D. Energy Accounting

Correct accounting for task and function energy in the application and system software is one of the challenging parts of the design of our simulation framework. The energy accounting mechanism of our simulator is task-based. In other words, we keep a separate energy balance sheet (called task energy balance sheet, or TEBS) for each task running in the simulator. This is illustrated in Fig. 3. At the beginning of kernel initialization, there is only one task, the *idle-task*. The name *idle-task* might be misleading since this task does a lot of the hardware initialization before spawning another task called *init-task*. The *init-task* continues with kernel software initialization, mounts a root filesystem near the end, and executes the first user-mode program, which can then spawn other application tasks. This sequence of events during the initialization is illustrated in Fig. 4.

The energy balance sheets in the simulator have a one-to-one correspondence with the tasks running in the simulator. At the beginning, there is only *idle-task*, so there is only one energy balance sheet. When the *init-task* is created, a new energy balance sheet is created and the energy value for the new balance sheet is initialized to zero. Anytime a context switch occurs, the simulator is able to detect the context switch non-intrusively and attribute the energy consumption of the embedded system to the running task accordingly. Therefore, at any point in time, the energy value in each energy balance sheet indicates the energy consumption of the embedded system due to the corresponding task. The sum of energy values from all the energy balance sheets equals the total energy consumption of the embedded system.

Energy profiling for individual software function is more involved. Basically, we maintain a *function energy* stack (FES) for each task. FES tracks the function call stack for the corresponding task. The only additional information stored in each slot of FES is the energy value of the task at the entry of a function. When the function exits, the current value of task energy minus the energy value at the entry is the energy consumption of that particular function instance. The slot is *closed* by storing the energy consumption value in a database.

We further illustrate the energy accounting mechanism with an example. Consider the scenario depicted in Fig. 5. In this scenario, there are two tasks in the system, namely *task 1* and *task 2*. The functions

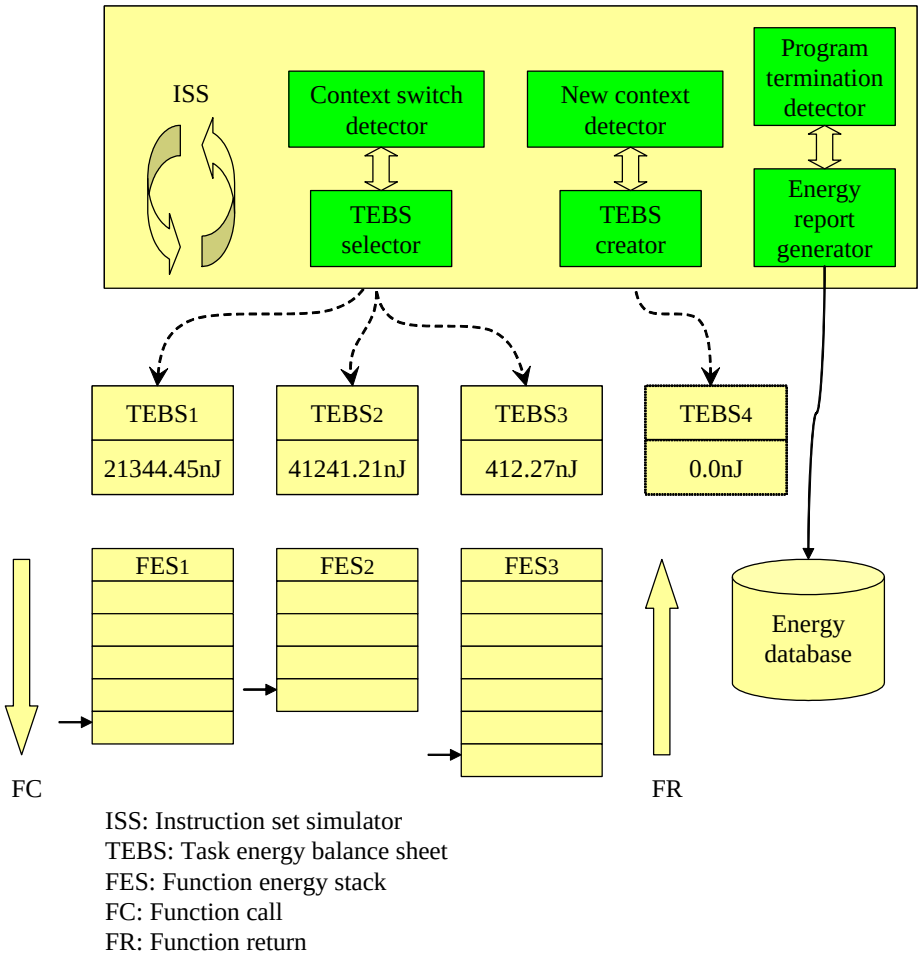

Fig. 3. Energy accounting illustration

contained in each task are also shown in Fig. 5. Each box represents a function. The tree structure connecting the boxes represents the function call hierarchy. For example, function `f1()` starts out by executing a sequence of instructions local to itself. It then calls function `f1a()`. When function `f1a()` returns, it calls function `f1b()`. When function `f1b()` returns, function `f1()` continues to execute more instructions local to itself, before coming to its end and returning to its parent function.

Fig. 6 shows the execution trace of this system starting at some point in time t_0, where the program counter is at the beginning of function `f1()`. The first two rows show the functions *task 1* or *task 2* are in at any point in time. Accordingly, associated with each task is a TEBS. Hence, the next two rows show the plots (TEBS1 and TEBS2) for the their respective TEBS's. During the time when *task 1* is executing, only the value of TEBS1 is increasing, whereas the value of TEBS2 stays unchanged. When *task 2* is executing (between t_7 and t_8), the value for TEBS1 stays unchanged, whereas the value of TEBS2 is increasing. Note that when context switch occurs at time t_7, *task 2* is entered at function `f2aa()`, because this is where it was switched out before time t_0.

The mechanism of function energy accounting is illustrated in the last two rows of Fig. 6. The FES's of *task 1* and *task 2* are selectively shown for some points in time (FES's for some points in time are not explicitly shown). A portion of the changes to the FES's is explained below:

1. At time t_0, function `f1()` is entered, and the TEBS1 value e_0 at this point in time is pushed into FES1.
2. At time t_1, function `f1()` calls function `f1a()`. The TEBS1 value e_1 at this point in time is pushed into FES1.
3. At time t_2, function `f1a()` calls function `f1aa()`. The TEBS1 value e_2 at this point in time is pushed

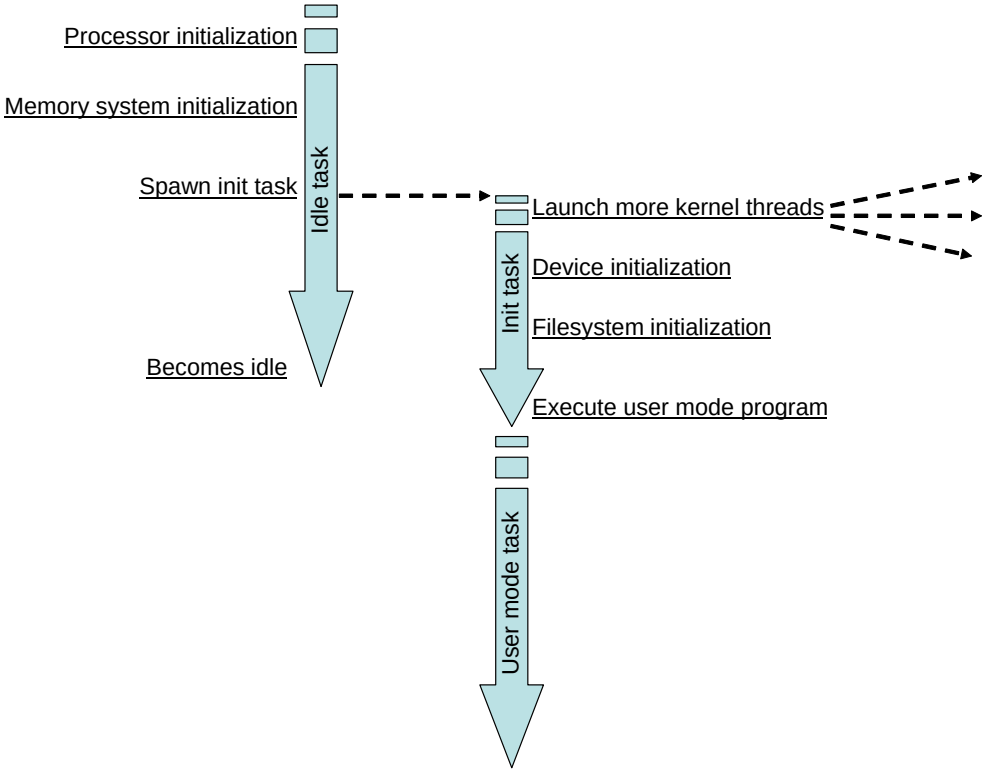

Fig. 4. Sequence of events during Linux OS initialization

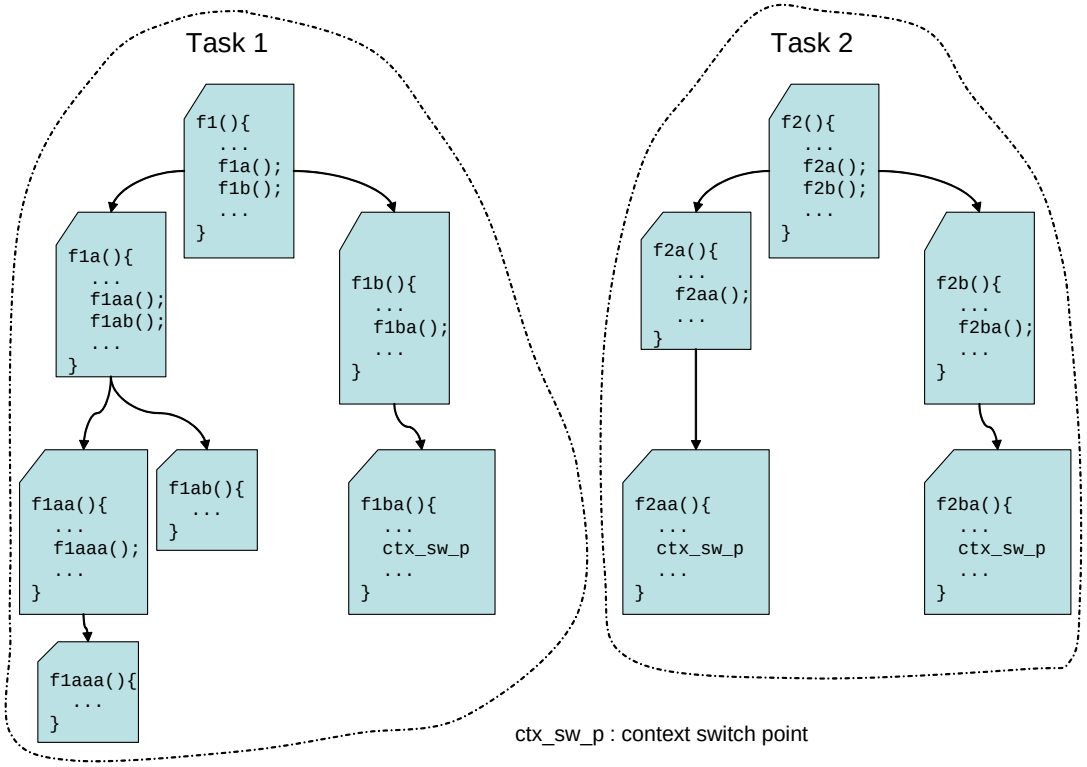

Fig. 5. A two task scenario used to illustrate the energy accounting mechanism

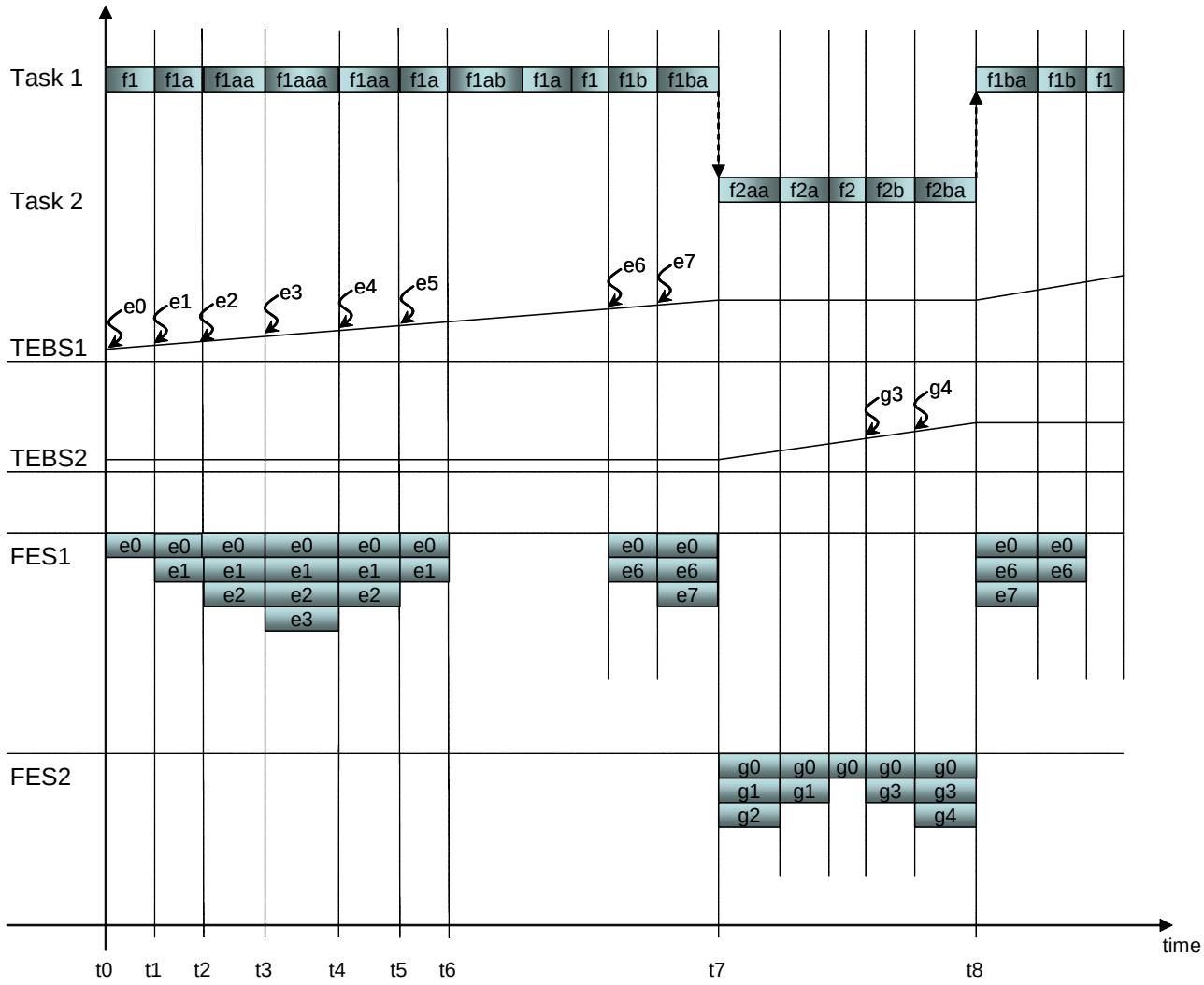

Fig. 6. Execution trace of the two-task system

into FES1. FES1 now contains three entries.

4. At time t_3, function `f1aa()` calls function `f1aaa()`. The TEBS1 value e_3 at this point in time is pushed into FES1. FES1 now contains four entries.
5. At time t_4, function `f1aaa()` returns. At this point in time, the TEBS1 value is e_4. The fourth entry in FES1 is closed, and the energy difference $e_4 - e_3$ is stored into the energy database as the energy consumption of the function instance `f1aaa()`.
6. At time t_5, function `f1aa()` returns. At this point in time, the TEBS1 value is e_5. The third entry in FES1 is closed, and the energy difference $e_5 - e_2$ is stored into the energy database as the energy consumption of the function instance `f1aa()`.
7. At time t_6, function `f1a()` returns. At this point in time, the TEBS1 value is e_6. The second entry in FES1 is closed, and the energy difference $e_6 - e_1$ is stored into the energy database as the energy consumption of the function instance `f1a()`.
8. When the context is switched to *task 2* at time t_7, all the subsequent FES operations are applied to FES2 instead.

Since task creation in the Linux OS is always accomplished by task cloning, FES is also cloned when a task is created. However, the entry energy values for all the slots in the new function energy stack are reset to zero, as is the energy value of the new energy balance sheet. This is necessary to avoid double counting.

The simulated program triggers the end of simulation by setting the program counter to 0x00000000.

Once this happens, the simulator starts consolidating the FES's for all the tasks in the simulator. Basically, all the slots in the FES's are closed and the *partial* energy consumptions of the functions are stored in the energy database. After that, the energy database is stored into a file for post-processing.

References

[1] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proc. Design Automation Conf.*, June 1998, pp. 576–581.

[2] T. Simunic, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Proc. Design Automation Conf.*, June 1999, pp. 867–872.

[3] A. Sinha and A. P. Chandrakasan, "JouleTrack - A web based tool for software energy profiling," in *Proc. Design Automation Conf.*, June 2001, pp. 220–225.

[4] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Trans. VLSI Systems*, vol. 2, no. 4, pp. 437–445, Dec. 1994.

[5] J. Flinn, K. I. Farkas, and J. Anderson, "Power and energy characterization of Itsy pocket computer (version 1.5)," Tech. Rep. TN-56, Western Research Laboratory, Feb. 2000.

[6] W. R. Hamburgen, D. A. Wallach, M. A. Viredaz, L. S. Brakmo, C. A. Waldspurger, J. F. Barlett, T. Mann, and K. I. Farkas, "Itsy: Stretching the bounds of mobile computing," *Computer*, vol. 34, no. 4, pp. 28–36, Apr. 2001.

[7] Intel Corporation, *Intel StrongARM SA-1100 Microprocessor Developer's Manual*, Aug. 1999.