Supporting System-Level Power Exploration for DSP Applications

Luca Benini [#] Marco Ferrero [‡] Alberto Macii[‡] Enrico Macii [‡] Massimo Poncino [‡]

[#] Università di Bologna
Bologna, ITALY 40136

[‡] Politecnico di Torino
Torino, ITALY 10129

Abstract

System-level power exploration requires tools for estimation of the overall power consumed by a system, as well as a detailed breakdown of the consumption of its main functional blocks. We focus on power estimation for data-dominated systems specified as synchronous data-flows and implemented on a single-processor architecture. Our estimator is integrated within the Ptolemy design environment, and provides information to system designers on the power dissipated by every task in a given specification. Power estimation is based on instruction-level power models. We demonstrate the applicability of our tool on a few design examples and target architectures.

1 Introduction

The increased productivity required for designing systems-on-chip mandates the availability of tools that can directly analyze, optimize, and synthesize system-level descriptions. This requirement represents a significant challenge, especially in the case of power-constrained designs like battery-operated systems, where the dynamic, workload-dependent nature of power dissipation further complicates the picture.

In the last few years, several works on high-level power estimation have appeared in the literature (see [1] for a survey). Most of these techniques addressed the register-transfer (RT) level and the behavioral level. System-on-chip and embedded system designers, however, need estimation tools to analyze complex descriptions with interacting software and hardware components. A major challenge in developing system-level estimation tools is the uncertainty in the definition of system-level descriptions.

A first class of approaches [2, 3] models the system as a set of *tasks* to be executed on a target processor. Interaction among tasks is described by a *task graph*, where nodes represent tasks, and edges represent dependencies. Periodic execution of the graph is assumed. The strength of these techniques is that they do not assume any specific hardware implementation, thus, they can be used to explore a wide design space. Two are the main disadvantages of this model. First, it is too abstract, because the task graph does not fully represent functional information about the tasks. Second, it emphasizes explicit intra-task communication and concurrency, leaving little room for algorithmic optimization and analysis.

A second class of approaches [4, 5, 6] relies on a structural description of the system, that is viewed as the interconnection of components (e.g., processor, memory, interconnect). System functionality is provided in algorithmic form (typically, an executable specification in an imperative language), and system design consists of optimally mapping functionality onto the architectural template. The accuracy of power estimation for such models may be comparable to that of RTL methods, since individual system-level components can be pre-characterized. The main limitation of these approaches is that they operate at a relatively low level of abstraction, where the degrees of freedom on hardware implementation are reduced. Furthermore, it may be hard to extract concurrency and perform high-level hardware-software partitioning starting from an algorithmic specification.

Some system-level design environments assume a specification style that lies somehow in between the previous two [7]. A system is still represented at an abstract level as a set of interacting tasks, yet the functional information is exposed and specified for each task. This specification style overcomes the limitations of traditional programming languages (such as C), which may be suited for general-purpose computational systems, but cannot easily handle the real-time, concurrent nature of embedded systems. To provide even more specification flexibility, the *Ptolemy* design environment [7] supports various *models of computation*, ranging from differential equations to discrete-event models, depending on the semantics of the domain being modeled. The main purpose of our work is to provide a power estimation backplane for this system-level specification style.

We focus on Ptolemy's *data-flow* computational model, that is particularly suited for describing signal processing systems. Data-flow semantics allows static scheduling of a description. In other words, a typical non-terminating signal processing task can be mapped onto a finite schedule that can be repeated indefinitely for infinite execution with bounded memory. Ptolemy provides algorithms for mapping an abstract data-flow specification onto a set of procedures which are executed on a target signal processor or C code is automatically generated, not only for the tasks, but also for the scheduler. The C code can then be compiled and executed on a target signal processor, or a general-purpose microprocessor.

Our tool is capable of estimating the energy and the time spent for the execution of each task and of the static scheduler. This information is automatically back-annotated into the system-level specification, and can be exploited by the designer to identify the tasks whose execution require most energy (or time), as well as the overhead due to the coordination of their execution. Our estimation engine closes the design exploration loop, which is left open in Ptolemy. In fact, it is now possible to explore alternative, functionally correct architectures, and choose the best one in terms of energy or execution time. Additionally, a designer can exploit detailed power and performance information to focus his/her effort on the most critical tasks.

An approach somehow close to ours has been proposed in [8]. Here, the focus is on Ptolemy's *discrete event* model of computation, which is well-suited for describing control-dominated reactive systems. The Ptolemy simulation environment is tightly coupled to a HW/SW partitioning tool to achieve concurrent power estimation of software and hardware components. This approach keeps the distinction between HW and SW models, and exploits the concurrent simulation capabilities of Ptolemy.

2 Ptolemy and Data-Flow Models

In Ptolemy, systems are modeled by sets of components connected through oriented arcs. Each component is a functional *block* which realizes a specific operation, at different abstraction levels: A block can implement a simple addition (atomic operation) as well as a complex FFT transform (non-atomic operation).
Blocks are named *stars* in Ptolemy. They can be hierarchically grouped because Ptolemy supports objects called *galaxies* which enclose sets of interconnected *stars*.
When Ptolemy is used to synthesize a source code description from a high-level specification, this block diagram is also named *program graph* because it is a visual description of that source code.
In general, a computational model specifies a set of rules (semantics) that govern the interaction of components. The *data-flow* is one of the supported computational models.
In a data-flow model, a program is divided into blocks which can execute (*fire*) whenever input data are available. These blocks communicate by sending messages through buffered FIFO channels in an asynchronous way. The evolution of the system consists thus of a proper sequence of firings that is called *schedule*. Depending on the constraints of the model, the schedule may be obtained *statically*, that is, by inspection of the data-flow network, or *dynamically*, that is, using run-time information.
In Ptolemy there are three types of data-flow models [9]:

- DDF (Dynamic Data-Flow): Firing sequences are generated at run-time because the number of samples read or written are data-dependent. Each block must specify the number of samples required at its inputs to activate the block next time. Due to its generality, this model slows simulation down significantly.
- BDF (Boolean-controlled Data-Flow): The number of samples that a process can read/write may be either a constant or a (Boolean) function of a control-sample read from a particular control-input. Static scheduling is not guaranteed; in that case, BDF models become DDF models.
- SDF (Synchronous Data-Flow): Firing sequences are determined once, during the start-up phase. The constraint is that a process can only read/write a constant number of samples. For this reason the firing order is periodic, making this model suitable for synchronous signal processing systems.

The three models of computation described above represent different tradeoffs between expressiveness and simplicity. DDF is the most expressive model (in fact, it is Touring-equivalent), but it is very hard to analyze and to prove properties on its execution (even testing for termination is undecidable in general DDFs). On the other hand, SDF cannot fully express control-dominated behavior (e.g., conditional execution), but it can be analyzed very efficiently. Not only termination can be tested efficiently, but it is possible to constructively build a schedule (i.e., an order of task execution) that requires only bounded channel buffers. BDF is between SDF an DDF for both generality and complexity.
In Ptolemy a *domain* defines a computational model. A domain must contain a specific object (*target*) that handles the execution inside the domain. This object, by means of a scheduler, defines a computational model and verifies the execution order of the functional blocks. Ptolemy provides two kinds of domains: The *simulation* domain and the *code generation* (CG) domain.
The code generation domain is used to synthesize code from a data-flow description. The CG domain uses either the BDF or the SDF computational model. All the code generation domains that are derived from the CG domain obey SDF semantics and can thus be scheduled at compile time.
A key feature of code generation domains is the notion of a target architecture. Every application must have a user-specified target architecture, selected from a set of targets supported by the user-selected domain. A CG domain supports either single-processor or multi-processor architecture. In this work we consider only the first one, and we focus on a specific target architecture, namely, the ARM processor.
The code fragments generated for each block are connected with the others through the code generated by the target. After the code is generated, the target calls the compiler and executes the code. This generation procedure is independent of the type of code generated.
The CG domain also fixes the target language. We focus on the domain that generates C code (CGC domain).

3 Power Exploration Support

The main steps required for power exploration support can be summarized as follows. From the C code generated by th CGC domain, we first generate executable code for the ARM processor using the standard ARM compiler. Then, we run the code on a modified instruction-level ARM simulator that estimates the power for the execution of each instruction on a cycle-by-cycle basis. Instruction-level power estimates are analyzed, and the total power spent in the execution of each task (and in the execution of the code required to support static scheduling) is obtained. Performance, expressed in terms of processor execution cycles, is monitored as well. This information is then back-annotated into the original specification. The user can then analyze and modify the specification to optimize power and/or performance.

3.1 Instruction-Level Power Analysis

We have extended the software emulator for the ARM processor (`ARMulator`) by implementing the instruction level power model described in [10]. `ARMulator` is an instruction level emulator that computes the number cycles needed by every instruction, without modeling the precise timing characteristics of the processor. `ARMulator` also supports a full ANSI C library to allow complete C programs, like those generated by Ptolemy, to run on the emulated systems.
We used the `ARMulator` available in the ARM Software Development Toolkit version 2.50. This emulator has a modular structure where various models can be connected to the core model, which emulates the behavior of the core of the ARM processors. Memory models, for example, are used to emulate the main memory system in an ARM-based configuration. In particular, they allow the specification of various access times for memories. Another model is the `Tracer` model, that can trace instruction execution and memory accesses of a program. We implemented our extensions to `ARMulator` inside the `Tracer` model.
The power costs of the instructions are read from a file during the initialization phase of the `Tracer` model. In this way, since `ARMulator` can emulate various processors, correct costs for the processor being emulated can be supplied to it. According to the terminology of [10], the only *inter-instruction* effects that we model are the pipeline stalls. They can be caused, for example, by a dependency between instructions in the pipeline, or by a reading or a writing with wait-states in the main memory. The pipeline is considered to be stalled only during a wait-state.

In this way, also cache-misses are considered because, in most cases, main memory access time is longer than processor cycle time, so during a cache-miss and subsequent line-fill, there are some wait-states. On the other hand, pipeline stalls caused by data-dependencies are not considered in this work because, usually, C compilers generate code already optimized in order to reduce the number of data-dependencies.
The various costs are read from the file and placed in memory as follows:

- The *base costs* are stored in a hash table. This table has an item for every instruction read from the file. The key is the mnemonic code of the instruction. The datum contains the base cost of the instruction and a numeric code, needed to identify the instruction when the *overhead costs* for that instruction are computed.
- The *overhead costs* are stored in a matrix; element (x, y) contains the overhead cost corresponding to a transition from instruction with numeric code x to instruction with numeric code y. If the *overhead cost* for a generic pair of instructions is not specified in the file, it is considered null.
- The average pipeline stall cost is stored in a scalar variable.

If the *base cost* of an instruction is not specified in the file, a *default base cost* is associated to that instruction. The *default base cost* is also specified in the power cost file. If one or both instructions for which the *overhead cost* is computed do not appear in the power cost file, this *overhead cost* is considered null.
It is important to emphasize that base costs also depend on the addressing mode of the instruction (e.g., register vs. immediate), and whether or not that instruction uses special hardware features of the ARM (e.g., the barrel shifter). However, in order to limit the slow-down of the emulator during the execution of a program, these second-order refinements are not accounted in the power cost we used, that considers a single *average base cost*.
Instruction costs are specified in terms of the supply current of the processor, therefore they are power quantities. To get the energy cost of an instruction, its power cost must be multiplied by the number of processor cycles required for its execution. Cycle count for an instruction is computed by counting the elapsed time (available through an internal function provided by `ARMulator`) from the beginning of the instruction execution. In formula, the energy cost for an instruction I (C_I) is computed as follows:

$$C_I = \begin{cases} C_B \cdot (\#\text{cycles} - \#\text{ws}) + C_{St} \cdot \#\text{ws} & \text{if } \#\text{ws} > 0 \\ C_B \cdot \#\text{cycles} + C_O & \text{if } \#\text{ws} = 0 \end{cases}$$

where C_B is the *power base cost*; C_{St} is the average *power pipeline stall cost*; C_O is the *energy overhead cost* between instruction I and the *previous* instruction; #cycles is the number of cycles needed for the execution of instruction I and #ws is the number of wait-states occurred during the execution of instruction I. If power costs are measured in mA, then energy costs must be measured in mA×#cycles. Notice that if a pipeline stall occurs, the *overhead cost* is not added to the instruction energy cost.
The sum of instruction energy costs for all instructions executed within a block of code gives the energy cost for that block. We can obtain the average power dissipated during execution of the block by dividing the energy cost by the number of cycles.

3.2 Interaction with Ptolemy

We are interested in estimating the energy required to execute each instance of code associated to one block (*star*). The energy required to execute the code of a *star* is obtained by adding the energies needed to execute *all* the instances of the code for that *star* in the program synthesized by Ptolemy.
The C program generated by Ptolemy is compiled to generate an executable image and a *listing* file where, for each C code line, the corresponding assembly codes are shown. Each C code line is copied with relative comments in the *listing* file, so we use this feature to detect the beginning of an instance of *star* code. The analysis of the *listing* file is carried out during the initialization phase of `ARMulator`. This analysis produces two tables:

- Table x has an item for each instance of the code of the *star* included in the generated program. The fields of this item are the name of the *star*, the line numbers at which each instance begins and ends in the C code, the address of the first instruction of the block, a Boolean variable which specifies whether or not that *star* has code associated with it, the cycle count and the energy count.

  Two instances of code for the same *star* have the same value in the *star* name field but typically different values in other fields. Star instances are identified by their position in this table. Numbering of instances in the table starts from one because code not associated with any *star* is identified with zero.
- Table y has an item for every address in the program corresponding to a boundary between two *star* code instances or between a *star* code instance and code which does not correspond to any *star*. Since the blocks that constitute a program are contiguous, the boundary address is the address of the first instruction in the next block. Each item of this table contains: The boundary address and a number to identify the *star* corresponding to the adjacent block. If the next block does not correspond to any *star*, it is labeled with zero. The various items are ordered in increasing order of boundary addresses.

Figure 1 shows the way a fragment of a program is mapped onto tables x and y. Numbers shown in square brackets in the code fragment specify the position of the item associated to the corresponding instance of *star* code in table x. The two code blocks which realize the loop are not associated to any *star* because they are introduced by the *target*, in order to realize a proper schedule of the *stars*. For this reason, elements associated to them in table y contain zero as identification code.

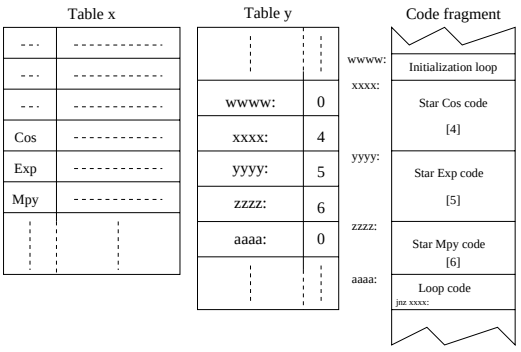

Figure 1: Code Fragment and Corresponding Tables.

The ARM C compiler normally produces code with the maximum optimization level. In this case, an instance of code associated to a *star* can be split into various code blocks. In this case table y would contain more items with the same identification code, as shown in Figure 2. In this way, table y has an item for every block in which a *star* code instance can be split, so various items in table y can refer to the same item in table x.

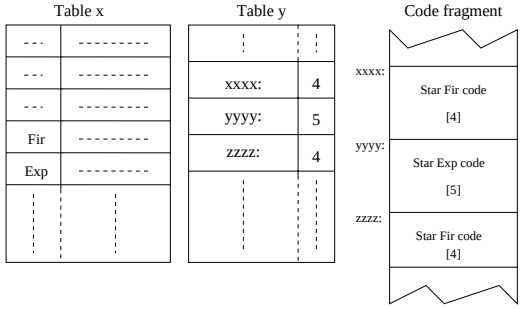

Figure 2: Star Code Instance Split in Two Blocks.

Addresses used in table y are the real addresses of instructions during program execution. They are computed by summing a fixed offset to the virtual addresses used in the *listing* file. These addresses, in fact, start from 0. The fixed offset is the result of the subtraction between the address of the first instruction in the *main* program of the executable and the address of this instruction in the *listing* file.
Once the instruction energy is computed with the method described in the previous subsection, it is summed to the corresponding variable containing the energy cost of the portion of program currently executed. A program generated by Ptolemy consists of the following parts:

- *Initialization code*. This code performs the various initializations and opens the files needed in the program. This code includes instructions introduced in transparent way by the linker and executed before the call to the *main* procedure.
- Instances of code associated to the various *stars*.
- *Scheduling code*. This code encapsulates the instances of *star* code and it performs the schedule of the *stars*. Schedule, in fact, is generated by Ptolemy before the real code generation phase.
- *Termination code*. This code performs final operations in the program and closes the files used by the program.

The various instances of *star* code are placed in *main*, so all addresses contained in table y refer to addresses in *main*. Energy costs due to execution of a function called in *main* are charged to the costs of the program sections (e.g., an instance of a *star* code) in which the function call occurs.
During the execution phase, if a code block corresponding to a *star* is executed, addresses of various executed instructions are compared to the address in the item that follows the item corresponding to the current code block in table y, in order to detect the starting point of the next contiguous code block. If the code executed does not belong to any *star*, the address of each executed instruction must be compared with all the addresses stored in table y because loops (`for` cycles) or other control structures (`if-then` or `do-while`) can introduce jumps to non-contiguous code blocks.

At the end of program execution, table x contains, for every instance of *star* code, the total number of cycles required to execute it, and the energy dissipated by the ARM during its execution. Dividing the energy consumed by the total number of cycles taken gives a measure of the *average* power dissipated by the ARM processor during the execution of that instance of the code. Initialization code, termination code, and scheduling code also have individual variables used to store time costs (in terms of number of cycles needed), and the energy cost. All this information is back-annotated into the *program graph* that is used as input by Ptolemy. This process of back-annotation is made during the execution of the program by the modified version of `ARMulator` described here.
Figure 3 shows the interface between Ptolemy and `ARMulator`, with the corresponding data and exchange formats. Ptolemy, besides using the ARM C compiler (i.e., `armcc`) to compile the C code and to generate the *listing* file, also calls an executable image analyzer, `decaxf`, and finally uses the modified version of `ARMulator` to get the back-annotation of the *program graph*.

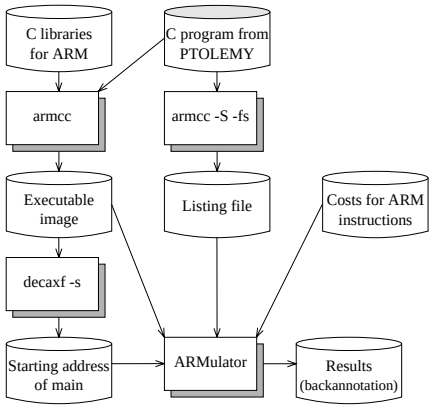

Figure 3: Interface between Ptolemy and ARMulator.

This process is done when Ptolemy recalls compilation of generated code. The single file generated by Ptolemy is shown at the top of the diagram.
`ARMulator` also provides a way of specifying the clock speed of the processor. This number is used to compute the number of wait-states that occur in a memory access. This allows to simulate programs with various main memory systems, each one with a different access time.
During execution, `ARMulator` does not perform I/O operations, in order to limit the overhead introduced by power characterization. Only at the end of this phase, results are written in a file for back-annotation purposes. In this way emulation speed of 112,000 ARM instructions per second have been obtained on a Sun `ULTRA10` workstation with 128 MB of RAM and processor working at 300 MHz.

4 System-Level Power Estimation Results

In this section we report a few examples on how our tool can be used to support design space exploration at the system level. The instruction-level power model described in Section 3.1 has been back-annotated with cost-values obtained for the ARM810 processor. These costs are obtained from [11] with supply voltage of 3.3 V and cycle time of 10 ns. The number of cycles required to execute a program is evaluated by computing the number of memory bus cycles; for this reason, the core must operate at the same frequency as the memory bus.

As a case study, we have compared two different implementations of a IIR (*Infinite Impulse Response*) filter. Such a filter can be described, in Ptolemy, using *stars* that belong to the CGC domain in two ways:

a) With a block that directly realizes the IIR function (see Figure 4 (a)); parameters of the block are the coefficients of the transfer-function and the gain of the filter. When activated, this block generates an output sample and consumes an input sample.

b) By connecting in a ring topology two blocks, which realize two FIR filters, with a unit delay on the feedback edge of the ring (see Figure 4 (b)). The ring is closed with a block that subtracts the delayed sample which comes from the feedback edge from the input sample. A block that multiplies the input samples by a value *gain* is connected to the ring's output. This block allows the tuning of the IIR filter gain.

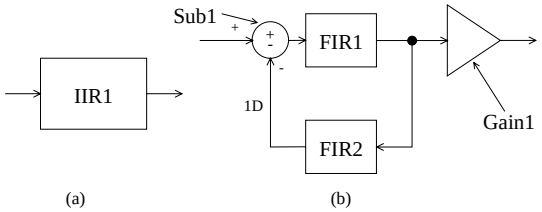

Figure 4: Two Realizations of an IIR Filter in Ptolemy.

Both implementations are fed by a stream which models a sampled sinusoidal waveform. The program graph describing the complete system obeys to the SDF semantics and its execution is managed by `default-CGC` *target*. This program graph is a DSP demo for the `CGC` domain included in Ptolemy. The program is synthesized and then executed by `ARMulator` assuming that the processor is an ARM810 and that there are no wait-states during the accesses to the main memory. Periodic schedule of *stars* is repeated 30 times; the C code implementing the operations of the *star* is then included in a `for` loop of 30 iterations.
Table 1 collects the results provided by `ARMulator` for the blocks of implementation (b) of the filter. Column *Time* gives the number of cycles taken by each block to finish the execution, while column *Energy* gives the energy (in mA×#cycles) dissipated by each block. Finally, column *Avg. Curr.* shows the average supply current for each block (in mA). This current is obtained by dividing the energy value by the number of cycles.

Star	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]
Sub1	8427	469205.5	55.68
FIR1	127135	7154634.5	56.28
FIR2	65189	3672611.8	56.34
Gain1	9561	544334.0	56.93
Total	210312	11840785.8	56.30

Table 1: Results for Solution (b), ARM810.

Table 2 compares the results for solution (a) to the ones for solution (b). In terms of energy dissipation, solution (b) is better. However, since solution (b) executes the same number of operations in a shorter time, the switching activity inside the processor is greater. For this reason, the average current value, drawn by the processor in solution (b), is higher than the current drawn in solution (a).

Solution	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]
(a)	222074	12439355.0	56.01
(b)	210312	11840785.8	56.30

Table 2: Results for 30 Iterations, ARM810.

If the activation frequency is the same for all the *stars*, the code of the schedule is realized by a simple `for` loop. This is the case of this example. Its execution requires only 565 cycles and its energy cost is 31131.5 mA×#cycles. As the complete execution of the program requires 2031944 cycles and 112985336.0 mA×#cycles, the overhead due to the schedule code is limited.
To confirm the results we have obtained, we have performed the following experiment. Instead of considering the ARM810 as target processor, we adopted the ARM7TDMI processor (which has no cache) and the ARM920T processor (which has a disjoint cache for data and instructions). In both cases, we have made the assumption that accesses to the main memory happen without wait-states.
Tables 3 and 4 show the results regarding the ARM7TDMI processor, while Tables 5 and 6 refer to the case of the ARM920T processor. As for the ARM810 architecture, solution (b) performs better than (a), in terms of energy consumption, for both the ARM7TDMI and the ARM920T.

Star	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]
Sub1	5354	302243.2	56.45
FIR1	79838	4653948.5	58.29
FIR2	41076	2397717.8	58.37
Gain1	5959	355324.2	59.63
Total	132227	7709233.7	58.30

Table 3: Results for Solution (b), ARM7TDMI.

Solution	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]
(a)	140244	8112749.0	57.85
(b)	132227	7709233.7	58.30

Table 4: Results for 30 Iterations, ARM7TDMI.

Star	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]
Sub1	4873	271930.3	55.80
FIR1	80715	4563778.0	56.54
FIR2	41536	2351503.8	56.61
Gain1	6119	350605.2	57.30
Total	133243	7537817.3	56.60

Table 5: Results for Solution (b), ARM920T.

Solution	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]
(a)	140496	7899399.0	56.23
(b)	133243	7537817.3	56.60

Table 6: Results for 30 Iterations, ARM920T.

In the next experiment, the ARM810 processor accesses the main memory with 1 wait-state for sequential addresses and 2 wait-states for non-contiguous addresses. Tables 7 and 8 report the results for this case. With respect to the previous tables, a further column is added (ΔE): It contains the increase, in percentage, of the energy dissipation due to the introduction of wait-states. The increase of energy dissipation is high and it is due to the growth of the number of cycles required by the program to execute. Each time a wait-state occurs, the pipeline is supposed to stall. Supply current drawn by the processor during pipeline stall is similar to the current drawn during normal instruction execution. All these facts confirm the increase of the dissipated energy during execution of the program.

Star	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]	ΔE [%]
Sub1	10876	595928.7	54.79	27.0
FIR1	168698	9381105.0	55.61	31.1
FIR2	86465	4815840.5	55.70	31.1
Gain1	12686	714426.9	56.32	31.2
Total	278725	15507301.1	55.64	31.0

Table 7: Results for Solution (b), ARM810 with W-S.

Solution	*Time* [#cycles]	*Energy* [mA×#cycles]	*Avg. Curr.* [mA]	ΔE [%]
(a)	294464	16319737.0	55.42	31.2
(b)	278725	15507301.1	55.64	31.0

Table 8: Results for 30 Iterations, ARM810 with W-S.

Also in this case, solution (b) is preferable. If the processor was not cached, the number of cycles required to execute each *star* code and, as a consequence, the corresponding energy cost would be even larger.

Notice that the difference, *in percentage*, between the energy dissipated by solution (a) and the one dissipated by solution (b) (around 5%) preserves almost the same value despite changing the emulated processor. This confirms that this kind of estimation is suitable to high level descriptions because it is not influenced by variations in the duration of the instructions. This is a *relative* estimation with the objective of driving the choice between various Ptolemy *program graph* solutions, corresponding to software design solutions.

5 Conclusions and Future Work

In order to help designers in building low-power systems, we need to provide detailed feedback on where and how power is dissipated. In this paper we focused our attention to the development of a flexible framework for system-level power estimation of signal-processing applications.

We integrated power estimation capabilities within the synchronous data-flow domain of the Ptolemy system-level design environment. Our estimators can provide feedback on the power consumed in performing the computation of each atomic block of a given SDF graph, when the target hardware architecture is a processor core. Power estimation is based on instruction-level simulation and power modeling.

We applied our estimation tool to a few system-level specifications; its flexibility was demonstrated by providing detailed task-by-task power estimates for functionally equivalent specifications mapped onto several different target cores.

Future work will focus on integrating memory and I/O power models within the instruction-level simulation backplane, in order to provide detailed power estimation not only on the power consumed by the processor, but also on the consumption of all its ancillary components.

References

[1] E. Macii, M. Pedram, F. Somenzi, "High-Level Power Modeling, Estimation and Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 11, pp. 1061-1079, November 1998.

[2] L. Benini, R. Hodgson, P. Siegel, "System-Level Power Estimation and Optimization," *ISLPED-98: ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 173-178, Monterey, CA, August 1998.

[3] D. Kirovski, C. Lee, M. Potkonjak, W. Mangione-Smith, "Synthesis of Power Efficient Systems-on-Silicon," *Asian and South Pacific Design Automation Conference*, pp. 557-562, February 1998.

[4] D. Lidsky, J. Rabaey, "Early Power Exploration: A World Wide Web Approach," *DAC-33: ACM/IEEE Design Automation Conference*, pp. 27-32, Las Vegas, NV, June 1996.

[5] T. Simunic, L. Benini, G. De Micheli, "Cycle-Accurate Simulation of Energy Consumption in Embedded Systems," *DAC-36: ACM/IEEE Design Automation Conference*, pp. 867-872, New Orleans, LA, June 1999.

[6] J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems," *DAC-35: ACM/IEEE Design Automation Conference*, pp. 188-193, San Francisco, CA, June 1998.

[7] B. Lee, T.M. Parks , "Dataflow Process Networks," *Proceedings of the IEEE*, pp. 773-799, May 1995.

[8] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, A. Sangiovanni-Vincentelli, "Efficient Power Estimation Techniques for HW/SW Systems", *IEEE Alessandro Volta Memorial Workshop on Low-Power Design,* pp. 191-199, Como, Italy, March 1999.

[9] B. Lee, D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, September 1987.

[10] V. Tiwari, S. Malik, A. Wolfe, M. Lee, "Instruction Level Power Analysis and Optimization of Software," *Journal of VLSI Signal Processing*, Vol. 13, No. 1-2, pp.223-233, 1996.

[11] P. Laramie, "Instruction Level Power Analysis and Low Power Design Methodology of a Microprocessor," *CS Master Thesis*, University of California, Berkeley.