Power Consumption Estimation of a C Program for Data-Intensive Applications

Eric Senn, Nathalie Julien, Johann Laurent, and Eric Martin

L.E.S.T.E.R., University of South-Brittany, BP92116
56321 Lorient cedex, France
{eric.senn, nathalie.julien, johann.laurent, eric.martin}@univ-ubs.fr
http://lester.univ-ubs.fr:8080/

Abstract. A method for estimating the power consumption of an algorithm is presented. The estimation can be performed both from the C program and from the assembly code. It relies on a *power model* for the targeted processor. Without compilation, several targets can be compared at the C-level in order to rapidly explore the design space. The estimation can be refined afterwards at the assembly level to allow further code optimizations. The power model of the Texas Instrument TMS320C6201 is presented as a case study. Estimations are performed on real-life digital signal processing applications with average errors of 4.2 % at the C-level, and 1.8 % at the assembly level.

1 Introduction

Power consumption is currently a critical parameter in system design. The fact is that the co-design step can lead to many solutions: there are many ways of partitioning a system, and many ways of writing the software even once the hardware is chosen. It is now well-known that the software has a very strong impact on the final power consumption [1]. To find the best solution is not obvious. Indeed, it is not enough to know whether the application's constraints are met or not; it is also necessary to be able to compare several solutions to seek the best one. So, the designer needs fast and accurate tools to evaluate a design, and to guide him through the design space. Without such a tool, the application power consumption would only be known by physical measurement at the very last stage of the design process. That involves to buy the targeted processor, the associated development tools and evaluation board, together with expansive devices to measure the supply current consumption, and that finally expands the time-to-market.

This work demonstrates that an accurate power consumption estimation can be conducted very early in the design process. We show how to estimate the power consumption of an algorithm directly from the C program without execution. Provided that the power model of the processor is available, there is no need for owning the processor itself, nor for any specific development tool, since it is not even necessary to have the code compiled. Thus, a fast and cheap comparison of different processors, or of different versions of an algorithm, is

possible [2]. In the following step of the design flow, a method for refining the estimation at the assembly level is also provided. This way, hot points can be definitely located in the code, and further optimizations can be focused on these critical parts.

Some power estimation methods are based on cycle-level simulations like in Wattch or SimplePower [3, 4]. However, such methods rely on a low-level description of the processor architecture, which is often unavailable for off-the-shelf processors. Another classical approach is to evaluate the power consumption with an instruction-level power analysis [5]. This method relies on current measurements for each instruction and couple of successive instructions . Its main limitation is the unrealistic number of measurements for complex architectures [6]. Finally, recent studies have introduced a functional approach [7, 9], but few works are considering VLIW processors [8]. All these methods perform power estimation only at the assembly-level with an accuracy from 4% for simple cases to 10% when both parallelism and pipeline stalls are effectively considered. As far as we know, only one unsuccessful attempt of algorithmic estimation has already been made [10].

Our estimation method relies on a *power model* of the targeted processor, elaborated during the *model definition* step. This model definition is based on the *Functional Level Power Analysis* of the processor architecture [12]. During this analysis, functional blocks are identified, and the consumption of each block is characterized by physical measurements. Once the power model is elaborated, the *estimation process* consists in extracting the values of a few parameters from the code; these values are injected in the power model to compute the power consumption. The estimation process is very fast since it relies on a static profiling of the code. Several targets can be evaluated as long as several power models are available in the library.

As a case study, a complete power model for the Texas Instruments TMSC6201 has been developed. It is presented in section 2 with details on the model definition. The estimation process, and the method to extract parameters from both the C and the assembly codes is exhibited in section 3. Estimations for several digital signal processing algorithms are presented in section 4. Estimations are performed both at the assembly and C-level, and compared with physical measurements. The gap between estimation and measure is always lower than 3.5% at the assembly-level, and than 8% at the C-level.

2 Model Definition

The model definition is done once and before any estimation to begin. It is based on a *Functional Level Power Analysis (FLPA)* of the processor, and provides the *power model*. This model is a set of consumption rules that describes how the average supply current of the processor core evolves with some *algorithmic* and *configuration parameters*. Algorithmic parameters indicate the activity level between every functional block in the processor (parallelism rate, cache miss rate

...). Configuration parameters are explicitly defined by the programmer (clock frequency, data mapping ...).

The model definition is illustrated here on the TI C6x. This processor was initially chosen to demonstrate the methodology on a complex architecture. Indeed, it has a VLIW instructions set, a deep pipeline (up to 11 stages), and parallelism capabilities (up to 8 operations in parallel). Its internal program memory can be used like a cache in several modes, and an External Memory Interface (EMIF) is used to load and store data and program from the external memory [11]. The use of an instruction level method for such a complex architecture would conduct to a prohibitive number of measurements.

The FLPA results for the TI C6x are summarized on the Figure 1. Three blocks and five parameters are identified. These parameters are called *algorithmic parameters* for their value actually depends on the algorithm. The parallelism rate α assesses the flow between the FETCH stages and the internal program memory controller inside the IMU (Instruction Management Unit). The processing rate β between the IMU and the PU (Processing Unit) represents the utilization rate of the processing units (ALU, MPY). The activity rate between the IMU and the MMU (Memory Management Unit) is expressed by the program cache miss rate γ. The parameter τ corresponds to the external data memory access rate. The parameter ε stands for the activity rate between the data memory controller and the Direct Memory Access (DMA). The DMA may be used for fast transfer of bulky data blocks from external to internal memory ($\varepsilon = 0$ if the DMA is not used).

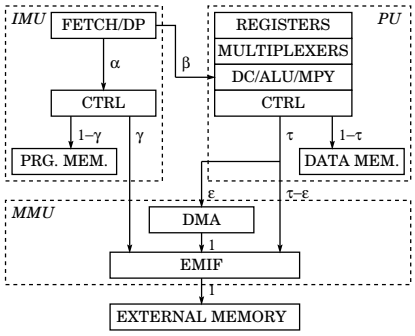

Fig. 1. FLPA for the C6x

To the former algorithmic parameters are added three *configuration parameters*, that also strongly impact on the processor's consumption: the clock frequency F, the memory mode MM, and the data mapping DM.

The influence of F is obvious. The C6x maximum frequency is *200 MHz*, but the designer can tweak this parameter to adjust consumption and performances.

The memory mode *MM* illustrates the way the internal program memory is used. Four modes are available. All the instructions are in the internal memory in the *mapped mode* (MM_M). They are in the external memory in the *bypass mode* (MM_B). In the *cache mode*, the internal memory is used like a direct mapped cache (MM_C), as well as in the *freeze mode* where no writing in the cache is allowed (MM_F). Internal logic components used to fetch instructions (for instance tag comparison in cache mode) actually depends on the memory mode, and so the consumption.

The data mapping impacts on the processor's consumption for two reasons. First, the logic involved to access a data in internal or in external memory is different. Secondly, whenever a data has to be loaded or stored in the external memory, or whenever two data in the same internal memory bank are accessed at the same time, the pipeline is stalled and that really changes the consumption.

Hence the final *power model* for the TI C6x, presented in the Figure 2.

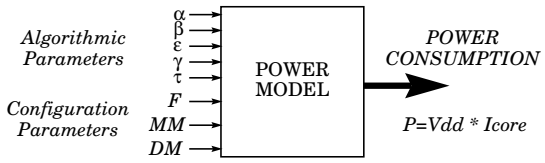

Fig. 2. Power Model

This model comes with a set of *consumption rules* that gives the power consumption of the processor, given the former parameters' values. To determine these rules, the parameters were made to vary, with the help of small assembly programs. Variations of the processor's core supply current were measured, and mathematical functions were obtained by curve fitting. For this processor, no significant difference in power consumption was observed between an addition and a multiplication, or a read and a write in the internal memory. Moreover, the effect of data correlation on the global power consumption appeared lower than 2%. More details on the consumption rules and their determination can be found in [12].

3 Estimation Process

To estimate the power consumption of a program with our power model, we must determine the value of all its input parameters. We will first explain how to precisely compute these parameter values from the assembly code, and then how to predict them directly from the C code without compilation.

3.1 Parameters Extraction from the Assembly Code

In the C6x, eight instructions are fetched at the same time. They form a *fetch packet*. In this fetch packet, operations are gathered in *execution packets* depending on the available resources and the parallelism capabilities. The parallelism rate α can be computed by dividing the number of fetch packet (NFP) with the number of execution packet (NEP) counted in the code. However, the effective parallelism rate is drastically reduced whenever the pipeline stalls. Therefore, the final value for α must take the number of pipeline stalls into account. Hence, a *pipeline stall rate* (PSR) is defined, and α is computed as follows:

$$\alpha = \frac{NFP}{NEP} \times (1 - PSR) \tag{1}$$

Identically, the PSR is considered to compute the processing rate β, with NPU the average number of processing unit used per cycle (counted in the code), and NPU_{MAX} the maximum number of processing units that can be used at the same time in the processor ($NPU_{MAX} = 8$ for the C6x):

$$\beta = \frac{1}{NPU_{MAX}} \frac{NPU}{NEP} \times (1 - PSR) \tag{2}$$

To determine the PSR, we must evaluate the number of cycles where the pipeline is stalled (NPS), and divide it by the total number of cycles for the program to be executed (NTC):

$$PSR = \frac{NPS}{NTC} \tag{3}$$

Pipeline stalls have several causes:

- a delayed data memory access: if the data is fetched in external memory (related to ε) or if two data are accessed in the same internal memory bank (related to the data mapping *DM*)
- a delayed program memory access: in case of a cache miss for instance (related to the cache miss rate γ), or if the cache is bypassed or freezed (related to the memory mode *MM*)
- a control hazard, due to branches in the code: we choose to neglect this contribution because only data intensive applications are considered.

As a result, NPS is expressed as the sum of the number of cycles for stalls due to an external data access NPS_τ, for stalls due to an internal data bank conflict NPS_{BC}, and for stalls due to cache misses NPS_γ.

$$NPS = NPS_\gamma + NPS_\tau + NPS_{BC} \tag{4}$$

Whenever a cache miss occurs, the cache controller, via the EMIF, fetch a full instruction frame (containing 8 instructions) from the external memory. The number of cycles needed depends on the memory access time T_{access}. As a result, with *NFRAME* the number of frames causing a cache miss:

$$NPS_\gamma = NFRAME \times T_{access} \tag{5}$$

Similarly, the pipeline is stalled during T_{access} for each data access in the external memory. That gives, with $NEXT$ the number of data accesses in external memory:

$$NPS_\tau = NEXT \times T_{access} \tag{6}$$

A conflict in an internal data bank is resolved in only one clock cycle. So, NPS_{BC} is merely the number of bank conflicts $NCONFLICT$.

$$NPS_{BC} = NCONFLICT \tag{7}$$

The three numbers $NEXT$, $NCONFLICT$ and $NFRAME$ can be computed from the assembly code. In fact, $NFRAME$ is also needed to compute the cache miss rate γ, with the total number of instruction frames in the code, and the cache size. However, since the assembly code for digital signal processing applications generally fits in the program memory of the C6x, γ is often equal to zero (as well as $NFRAME$).

The numbers $NEXT$ and $NCONFLICT$ are directly related to the data mapping. This mapping is expressed in the power model through the configuration parameter DM.

The number of DMA accesses can be counted in the assembly code. The DMA access rate ε is computed by dividing the number of DMA accesses by the total number of data accesses in the program.

At last, external data accesses are fully taken into account through $NEXT$ which includes the parameter τ; indeed, τ does not appear explicitly in our set of consumption rules.

3.2 Parameters Prediction from the C code

In the previous section, the parameters needed to actually estimate the power consumption of an application were extracted from the assembly code. In this section, we show how to determine these parameters directly from the C code, without compilation.

As stated before, the pipeline stall rate PSR is needed to compute the values of the parameters α and β. To calculate the PSR, we need the number of external data accesses $NEXT$, the number of internal data bank conflicts $NCONFLICT$, and the number of instruction frames that involve cache misses $NFRAME$ (Equations 3-7).

It is remarkable that the two numbers $NEXT$ and $NCONFLICT$ can be determined directly from the C program. Indeed, they are related to the data mapping which is actually fixed by the programmer (by the mean of explicit compilation directives associated to the C sources) and only taken into account by the compiler during the linkage. Accesses to the DMA are explicitly programmed as well. Because the programmer knows exactly the number of DMA accesses, he can easily calculate the DMA access rate ε without compilation.

Nevertheless, it is not possible to predict *NFRAME* at the C-level. Indeed, the assembly code size is needed to be compared with the cache size; a compilation is necessary. As explained before, the C6x cache is however large enough for most of the digital signal processing applications, and in these cases *NFRAME* and γ equal zero. Whenever *NFRAME* and γ are not known in the early step of the design process, it is still possible to provide the designer with *consumption maps* to guide him in the code writing [2].

To determine α and β at the C-level, the three parameters NFP, NEP and NPU must be predicted from the algorithm (instead of being counted in the assembly code). It is clear that this prediction must rely on a model that anticipates the way the code is executed on the target. According to the processor architecture and with a little knowledge on the compiler, four *prediction models* were defined:

The *sequential model* (SEQ) is the simplest one since it assumes that all the operations are executed sequentially. This model is only realistic for non-parallel processor.

The *maximum model* (MAX) corresponds to the case where the compiler fully exploits all the architecture possibilities. In the C6x, 8 operations can be done in parallel; for example 2 loads, 4 additions and 2 multiplications in one clock cycle. This model gives a maximum bound of the application consumption.

The *minimum model* (MIN) is more restrictive than the previous model since it assumes that load and store instructions are never executed at the same time - indeed, it was noticed on the compiled code that all parallelism capabilities were not always fully exploited for these instructions. That will give a reasonable lower bound for the algorithm's power consumption.

At last, the *data model* (DATA) refines the prediction for load and store instructions. The only difference with the MAX model is to allow parallel loads and stores only if they involve data from different memory banks. Indeed, there are two banks in the C6x internal data memory, which can be accessed in one clock cycle.

Assuming data-intensive applications, the prediction is performed by applying those models for each significant loop of the algorithm. Operations at the beginning or at the end of the loop body are neglected. As illustration, we present below a simple code example:

```
For (i=0; i<512; i++) {Y=X[i]*(H[i]+H[i+1]+H[i-1])+Y;}
```

In this loop nest, there are 4 loads (LD), and 4 other operations (OP): 1 multiplication, and 3 additions. In our example, the final store for Y, only done once at the end of the loop, is not considered. Here, our 8 operations will always be gathered in one single fetch packet so $NFP = 1$. Because no NOP operation is involved, $NPU = 8$ and α and β parameters have the same value. In the *SEQ* model, instructions are assumed to be executed sequentially. Then $NEP = 8$, and $\alpha = \beta = 0.125$. Results for the other models are summarized in Table 1.

Of course, realistic cases are more elaborated: the parameter prediction is done for each part of the program (loops, subroutines ...) for which local values

Table 1. Prediction models for the example

model	EP1	EP2	EP3	EP4	$\alpha = \beta$
MAX	2LD	2LD,4OP	-	-	0.5
MIN	1LD	1LD	1LD	1LD,4OP	0.25
DATA	2LD	1LD	1LD,4OP	-	0.33

are obtained. The global parameters values, for the complete C source, are computed by a weighted averaging of all the local values. Such an approach permits to spot "hot points" in the program. In the case of data-dependent algorithms, a statistic analysis should be performed to get those values.

4 Application

The estimation at the assembly level was already validated by direct comparison with measurements in [12]. In this section, the same process is applied at the C-level, and the two approaches are finally compared.

The estimation is performed for several digital signal processing algorithms: a FIR filter, a FFT, a LMS filter, a Discrete Wavelet Transform (DWT) with two different image sizes, an Enhanced Full Rate (EFR) Vocoder for GSM, and a MPEG1 Decoder (1600 lines in the C program; 4000 lines in the assembly code). The results are presented for different memory modes (mapped, cache and bypass) and data mappings (*EXT*ernal or *INT*ernal memory).

In Table 2, the value of the power model parameters extracted from the assembly code, and from the C code assuming the DATA prediction model, are presented. For these applications, $\gamma = 0$ since all the code is contained in the internal program memory, and $\varepsilon = 0$ since the DMA is not used. The PSR measured value (PSR^m), obtained with the help of the TI development tool, is used for estimation at the assembly level (but the calculated value could be used as well). The average error between the estimated (PSR) and the measured (PSR^m) pipeline stall rates is 3.2%. It never exceeds 5.5% which indicates the PSR estimation accuracy.

The power consumption of the algorithm is computed from those parameters' values. The relative error between estimation and measurements is given in Table 3. Results are given for the assembly level and for the four prediction models at the C-level.

Of course, the SEQ model gives the worst results since it does not take into account the architecture possibilities (parallelism, several memory banks etc.). In fact, this model has been developed to explore the estimation possibilities without any knowledge about the architecture of the targeted processor. It seems that such an approach cannot provide enough accuracy to be satisfying.

It is remarkable that, for the LMS in bypass mode, every model overestimates the power consumption with close results. This exception can be explained by the fact that, in this marginal memory mode, every instruction is loaded from

Table 2. Parameters estimations ($F = 200MHz$)

	Configuration		Assembly level			C-level		
Application	*MM*	*DM*	α	β	PSR^m	α	β	PSR
FIR	MM_M	INT	0.492	0.454	0	0.5	0.5	0
FFT	MM_M	INT	0.099	0.08	0.64	0.119	0.113	0.604
LMS-1	MM_B	INT	-	0.029	0.93	-	0.0312	0.95
LMS-2	MM_C	INT	0.625	0.483	0.25	0.76	0.475	0.24
DWT-1 (64*64)	MM_M	INT	0.362	0.287	0.0027	0.365	0.324	0.0269
DWT-2 (64*64)	MM_M	EXT	0.0915	0.0723	0.755	0.105	0.0932	0.713
DWT-3 (512*512)	MM_M	EXT	0.088	0.0695	0.765	0.1	0.089	0.726
EFR	MM_M	INT	0.594	0.472	0.225	0.669	0.479	0.219
MPEG	MM_M	INT	0.706	0.715	0.108	0.682	0.568	0.09

the external memory and thus pipeline stalls are dominant. As the SEQ model assumes sequential operations, it is the most accurate in this mode.

For all the other algorithms, the MAX and the MIN models always respectively overestimates and underestimates the application power consumption. Hence, the proposed models need a restricted knowledge on the processor architecture; but they guaranty to bound the power consumption of a C algorithm with reasonable errors.

Table 3. Power estimation vs measurements

	Measurements	Estimation vs Measure (%)				
Application	$P(W)$	*Asm*	*SEQ*	*MAX*	*MIN*	*DATA*
FIR	4.5	2.3	-38	5.5	-24.3	5.5
FFT	2.65	2.5	-10	28.5	-1	2.87
LMS-1	4.97	3.5	1.4	2.8	2	2.8
LMS-2	5.66	-1.8	-50	6.4	-15.2	6.4
DWT-1	3.75	1.9	-27	4.7	-13.2	4.7
DWT-2	2.55	-0.2	-10	3.4	-4.2	3.4
DWT-3	2.55	-1	-10.4	2.4	-4.7	2.4
EFR	5.07	-2.8	-50	11.1	-24	1.5
MPEG	5.83	0.7	-54	10	-33	-8
Average errors:		1.8	27.8	8.3	-13.5	4.2

The DATA model is the more accurate since it provides a maximum error of 8 % against measurements. After compilation, the estimation can be performed at the assembly level where the maximum error is decreased to 3.5%.

5 Conclusion

The main interest in this work is to propose an accurate and fast estimation of a C program without compilation. This estimation relies on a prediction that includes parallelism capabilities and pipeline stalls, which strongly impact on the power consumption. The method is therefore suitable to complex processors. Whenever the compiled code is too large for the target's internal program memory, the cache miss rate γ is hardly predictable. In this case, consumption maps are proposed, to summarize the variations of the power consumption [2].

Current works include the development of an on-line tool and the extension of the power models library to other processors. Future works will address the prediction of the execution time from the algorithm, to also achieve energy estimation at the C-level.

References

1. M. Valluri, L. John "Is Compiling for Performance == Compiling for Power?," presented at the 5th Annual Workshop on Interaction between Compilers and Computer Architectures INTERACT-5, Monterey, Mexico (2001)
2. N. Julien, E. Senn, J. Laurent, E. Martin "Power Consumption Estimation of a C Algorithm: A New Perspective for Software Design", in Proc. of the Sixth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, ACM LCR'02 (2002)
3. D. Brooks, V. Tiwari, M. Martonosi "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations" in Proc ISCA (2000)
4. W. Ye, N. Vijaykrishnan, M. Kandemir, M.J. Irwin "The Design and Use of SimplePower: A Cycle Accurate Energy Estimation Tool" in Proc. Design Automation Conf. (2000)
5. V. Tiwari, S. Malik, A. Wolfe "Power analysis of embedded software: a first step towards software power minimization" IEEE Trans. VLSI Systems, vol.2 (1994)
6. B. Klass, D.E. Thomas, H. Schmit, D.F. Nagle "Modeling Inter-Instruction Energy Effects in a Digital Signal Processor," presented at the Power Driven Microarchitecture Workshop in ISCA (1998)
7. S. Steinke, M. Knauer, L. Wehmeyer, P. Marwedel "An accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations," in Proc. PATMOS (2001)
8. L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria, R. Zafalon "A Power Modeling and Estimation Framework for VLIW-based Embedded Systems," in Proc. PATMOS (2001)
9. G. Qu, N. Kawabe, K. Usami, M. Potkonjak "Function-Level Power Estimation Methodology for Microprocessors," in Proc. Design Automation Conf. (2000)
10. C. H. Gebotys, R. J. Gebotys "An Empirical Comparison of Algorithmic, Instruction, and Architectural Power Prediction Models for High Performance Embedded DSP Processors," in Proc. ACM Int. Symp. on Low Power Electronics Design (1998)
11. TMS320C6x User's Guide, Texas Instruments Inc. (1999)
12. J. Laurent, E. Senn, N. Julien, E. Martin "High Level Energy Estimation for DSP Systems," in Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS (2001)