A Hybrid Energy Estimation Technique for Extensible Processors

Yunsi Fei, *Student Member, IEEE*, Srivaths Ravi, Anand Raghunathan, *Senior Member, IEEE*, and Niraj K. Jha, *Fellow, IEEE*

Abstract—In this paper, we present an efficient and accurate methodology for estimating the energy consumption of application programs running on extensible processors. Extensible processors, which are getting increasingly popular in embedded system design, allow a designer to customize a base processor core through instruction set extensions. Existing processor energy macro-modeling techniques are not applicable to extensible processors, since they assume that the instruction set architecture as well as the underlying structural description of the micro-architecture remain fixed.

Our solution to the above problem is a hybrid energy macro-model suitably parameterized to estimate the energy consumption of an application running on the corresponding application-specific extended processor instance, which incorporates *any* custom instruction extension. Such a characterization is facilitated by careful selection of macro-model parameters/variables that can capture both the functional and structural aspects of the execution of a program on an extensible processor. Another feature of the proposed energy characterization flow is the use of regression analysis to build the macro-model. Regression analysis allows for in-situ characterization, thus allowing arbitrary test programs to be used during macro-model construction.

We validated the proposed methodology by characterizing the energy consumption of a state-of-the-art extensible processor (Tensilica's Xtensa). We used the macro-model to analyze the energy consumption of several benchmark applications with custom instructions. The mean absolute error in the macro-model estimates is only 3.3%, when compared to the energy values obtained by a commercial tool operating on the synthesized register-transfer level (RTL) description of the custom processor. Our approach achieves an average speedup of three orders of magnitude over the commercial RTL energy estimator. Our experiments show that the proposed methodology also achieves good relative accuracy, which is essential in energy optimization studies. Hence, our technique is both efficient and accurate.

I. INTRODUCTION

In embedded system design, high silicon efficiency is required to meet tight cost, area, timing and power constraints. At the same time, programmability or customizability is also desired to augment or enhance system design in response to user specification, market change, or other rapidly evolving requirements. Various implementation options for system design, ranging from software running on a general-purpose programmable processor to custom hardware tuned for a specific application, exist and provide differing degrees of flexibility and efficiency. The recent availability of successful extensible processors promise a favorable trade-off between high efficiency (as seen in application-specific integrated circuits (ASICs)) and high flexibility (as seen in general-purpose processors), while keeping design turnaround times short. An extensible processor allows the designer to extend the instruction set of a base processor core through application-specific (custom) instructions. Thus, extensible processors can benefit embedded system design with their ability to simultaneously tune both the underlying hardware and the application software to meet diverse design requirements.

While commercial vendors of extensible processors, such as [1], [2], [3], [4], do offer design tools to take extensible processors from specification to hardware implementation, a large number of issues remain unresolved. One such open problem is energy estimation for extensible processors, which needs to be efficiently addressed since low power dissipation is a pre-requisite for most embedded systems. Recent work [5] has focused on providing the infrastructure to automatically select parts of an application that are best implemented using custom instructions based on performance metrics. Taking energy consumption of an application as the design metric in this scenario is challenging, which requires fast and accurate energy estimation for each candidate configuration. Note that if the extensions to the base processor instruction set architecture (ISA) have been decided already, a new energy estimation technique is not required, since any existing processor energy estimation/power analysis framework can be used to characterize the extended processor and estimate the energy consumed by an application. However, such an approach is impractical for use in energy optimization studies done in an application-specific instruction set processor (ASIP) design cycle, since energy characterization has to be performed for each configuration/extension, and the number of configurations/extensions is very large. In [6], a mathematical decomposition model is proposed for estimation of speed, area,

This work was supported by DARPA under contract no. DAAB07-00-C-L516.

Y. Fei and N. K. Jha are with the department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: yfei@ee.princeton.edu; jha@ee.princeton.edu).

S. Ravi and A. Raghunathan are with the NEC Labs America, Inc., Princeton, NJ 08540 USA (e-mail: sravi@nec-labs.com; anand@nec-labs.com).

and power of parameterizable soft intellectual property (IP). It decomposes the power function with respect to technology-dependent variables, presence or absence of clock gating (a power management technique), and architectural configuration variables, while accounting for the impact of architectural variables by making power dissipation proportional to area. In our paper, we focus on deriving the energy consumption of extensible processor with respect to the technology-independent variables, such as architectural and microarchitectural configurations.

Although existing processor energy estimation techniques are not directly applicable to the problem of efficient energy estimation for extensible processors, they offer valuable insights that can aid in the development of a good solution. Macro-modeling, which we adopt in this work, is a commonly used technique in processor energy estimation. It formulates the energy consumed by the processor in terms of parameters that are easily observable (say, during instruction set simulation). Two categories of processor macro-modeling techniques have been successfully employed. Structural macro-modeling approaches express the overall energy consumption in terms of the energy consumption of its constituent hardware blocks, and use the activity statistics of the hardware blocks for a given program trace to estimate energy. Instruction-level macro-modeling approaches, on the other hand, characterize the energy consumption of processor instructions using carefully constructed test programs and can use fast instruction set simulation to yield efficient energy estimates. Thus, structural approaches offer the benefits of high accuracy especially if they model the structure of the processor at a fine granularity, while instruction-level approaches facilitate fast energy estimation since they do not have to be cycle-accurate or structure-aware. In the case of extensible processors, which have a fixed base processor core and customizable components (due to instruction set extensions), we hypothesize that a hybrid approach that combines the efficiency of instruction-level approaches with the accuracy of structural approaches is best suited.

A. Paper Overview and Contributions

Our methodology involves deriving a composite energy macro-model by characterizing the energy consumed by applications with custom instructions using (i) instruction-level parameters that capture the interplay of the dynamic execution trace of a program and the base processor micro-architecture (inclusive of processor pipeline stalls and other effects, cache misses, *etc.*), and (ii) structural parameters that account for the energy effect of each instruction (base/custom) on the custom hardware. By using both instruction-level and structural parameters in the macro-model, and using instruction set simulation of an application with custom instructions to capture both instruction-level statistics as well as custom hardware usage data, both efficiency and accuracy can be simultaneously targeted.

A significant feature of our macro-modeling flow is that characterization is performed using *regression macro-modeling*, which has the following advantages.

- Variables in a regression macro-model can be chosen from instruction-level or structural domains, or both. Thus, regression macro-modeling is naturally applicable to our hybrid formulation.
- Regression macro-modeling significantly simplifies the process of constructing test applications or programs used in characterization. Conventional instruction-level approaches perform bottom-up macro-modeling, which requires test programs that contain isolated instructions, selected instruction sequences, *etc.*, wrapped in loops, in order to infer the average energy consumption of a given instruction under various scenarios. However, test program construction becomes cumbersome in most cases (for example, if test programs need to target instructions such as branch). Regression macro-modeling, through its in-situ characterization, only requires that the test programs have diversity in their instruction statistics and custom hardware instantiation so as to cover the instruction space and custom hardware library. Thus, arbitrary test programs can be used for regression macro-modeling.
- Construction and use of regression models are efficient, and the tools for building a regression model are widely available.

With the energy macro-model of the extensible processor built in the above manner, energy consumption of an application incorporating any custom instructions can simply be determined by instruction set simulation to capture instruction-level execution statistics, and dynamic resource usage analysis to derive custom hardware activation data needed by the macro-model. Note that energy estimation with this energy macro-model only needs the custom instructions descriptions, it does not require the custom processor to be synthesized. Thus, our methodology is easily usable for evaluating energy/performance vs. area trade-offs among different candidate custom instructions at the early design stage. To our best knowledge, this is the first and only work on energy estimation for extensible processors that can be embedded in the design cycle. We applied the proposed methodology to characterize the Xtensa extensible processor core from Tensilica Inc. [1]. We then used the energy macro-model of the Xtensa processor to evaluate the energy consumption of several applications with custom instructions. Our experimental results show that the mean absolute error in the macro-model estimates, when compared to the energy values computed by a commercial tool operating on the actual hardware description of extended processors, is only 3.3%, while the average speedup is three orders-of-magnitude.

B. Related Work

Various techniques have been developed to estimate and optimize power or energy consumption of hardware throughout its design hierarchy [7], [8]. Recently, attempts have also been made for energy estimation and optimization of software running on embedded processors. As mentioned before, these approaches can be classified, based on the macro-modeling employed, into *structural* and *instruction-level* techniques.

Structural techniques for energy estimation of software utilize the architectural description of the processor to collect

the dynamic activity information for each architectural block using simulation, calculate the energy consumption for each component, and finally sum them up to compute the overall energy consumption. Early work [9] characterized the power consumption of each architectural block as a single number. The power profiler in [10] calculates the energy consumption of functional units based on the switching activity between consecutive cycles. Wattch [11] and SimplePower [12] estimate the energy consumption at each cycle at the architecture level. Commercial tools such as WattWatcher from Sente [13] can also be used for energy estimation, once the RTL hardware description of the processor and the binary image of the program become available. However, RTL simulation of a processor is extremely slow for even small programs and methods for reducing the simulated trace become necessary [14].

Instruction-level macro-modeling techniques compute the energy consumption of a program based on its instruction profile. They primarily rely on the energy consumption characterization of each instruction of the processor and also estimate the energy consumption of special cases (such as cache misses) that can occur during the execution of a program. Characterization of each instruction can be performed by actual current measurements for a processor chip executing carefully created test programs [15]. The techniques in [16] measure the instantaneous processor power to build a software energy estimation model. The accuracy of instruction-level modeling is improved further by the techniques in [17], [18], [19], [20], which are cognizant of variations due to instruction encoding, addressing mode, register fields, operands values, bit toggling on internal and external busses, *etc.* Since the added accuracy comes at the cost of additional CPU time, efficiency is targeted in [18], [21] which perform measurements only on a subset of the instructions (for base energy) and instruction sequences (for inter-instruction effects). Measurement-based approaches are accurate because data are acquired from an actual chip implementation. The same reason, however, makes measurement based techniques infeasible for power trade-off studies early in the design cycle, especially if the hardware architecture is not fixed.

Recently, statistical analysis has also been used to build energy/power prediction models for very large instruction word (VLIW) and reduced instruction set computer (RISC) processors. The energy coefficients in the macro-model are not obtained through simulation or measurement, instead, they are calculated with *a priori* knowledge of energy characteristics of a set of instructions and the relevant statistical execution information of these instructions. Instruction-level functional approaches, such as [21], [22], [23], decompose an instruction into its constituent pipeline functions (for example, fetching and decoding, execution, load and store, write back, *etc.*), and calculate the energy coefficients for these functions. Structural models, such as [24], have variables corresponding to the fraction of total instructions executed by each architectural block. The technique proposed in [23] examines instructions with a finer granularity by considering, for example, instruction fetch address, instruction bit encoding, register numbers and immediates, data values, *etc.* However, these approaches do not provide an overall energy estimation for an application.

There has been only one previous work in the literature targeting the power estimation for ASIPs [25]. It requires the hardware description language (HDL) description of the processor, and hence is expensive to use in the design cycle. It first performs instruction set simulation to extract data type information, then connects it with the components used by each instruction and characterizes the power consumption of each component with various data types. After program profiling, power consumption for the application program is obtained. It aims for accuracy at the RTL and efficiency at the instruction level. Our approach has the same purpose. However, we target extensibility in the early design cycle of custom processor design through a hybrid approach of structural and instruction-level macro-modeling.

The rest of this paper is organized as follows. Section II briefly examines the Xtensa processor core used in this work. Section III examines with a motivational example the energy macro-modeling requirements for an extensible processor. Section IV describes the proposed energy estimation methodology and details the salient steps. Section V presents the results of applying the proposed methodology to build the energy macro-model of the Xtensa processor core and using it to evaluate the energy consumption of applications with different custom instructions. Finally, Section VI concludes.

II. THE EXTENSIBLE XTENSA PROCESSOR

We use the extensible Xtensa processor from Tensilica [1] as the target processor for macro-modeling and energy estimation. Extensible processors try to combine advantages of a general-purpose processor's flexible control logic and an ASIC's efficient computation part. Xtensa's ISA consists of a basic set of instructions which exists in all Xtensa implementations, plus a set of configurable and extensible options [26].

The base ISA defines approximately 80 instructions, and the basic hardware implementation of the Xtensa core is built around a traditional five-stage RISC pipeline, with a 32-bit address space. The configurable options include a wide range of architectural settings. For example, the designer can configure the base processor to include generic instructions (*e.g.*, multiply-accumulate) or floating point co-processors, customize the memory/cache architecture and register file, and set up interruption/exception mechanisms and levels. Extensibility is achieved by specifying application-specific functionality through custom instructions (also called Tensilica Instruction Extension or TIE).

The behavior of the custom instructions is described using a subset of the Verilog HDL. Custom instructions can be used to perform complex computations, which can take multiple clock cycles to complete. In instruction encoding, there are at most two input and one output operand fields in the instruction. If the custom computation needs additional inputs and outputs, custom state registers are added to allow instructions to have more sources and destinations. They can also be used as dedicated registers holding values for some temporary variables during program execution. Custom instructions can access both the general-purpose register file of

the base processor and additional custom registers/register files for their computations. The TIE compiler processes the custom instruction specification and facilitates seamless integration of the added custom hardware with the base processor configuration. The custom hardware extension consists of designer-specified functional units (corresponds to Verilog-like operators and custom built-in modules), storage elements (register files and custom state registers), pipeline flip-flops and control signals. Control logic, such as the TIE instruction decoder, bypass logic, interlock detection, and immediate-generation logic required by the custom instructions are automatically generated. After the custom instructions are incorporated, a processor generator automatically generates the enhanced processor, and the corresponding GNU-based software development kit for the configuration, which includes ANSI C/C++ compiler, linker, assembler, debugger, code profiler, cycle-accurate instruction set simulator (ISS), diagnostics, test benches, and standard libraries. Instead of invoking custom instruction at the assembly level, calls to custom instructions can be directly inserted into a high-level language (*e.g.*, C, C++) description of the application program. In this way, both the hardware and software are tuned for specific applications.

III. EXTENSIBLE PROCESSOR ENERGY MACRO-MODEL REQUIREMENTS

In this section, we illustrate with an example the different factors which must be considered in building an energy macro-model for an extensible processor, and identify the macro-model components.

A. Motivational Example

Example 1: Fig. 1(a) shows a portion of the architecture of an extended processor, wherein the base processor datapath has been augmented with custom hardware needed to implement three custom instructions: *MULT*, *MAC* and *CUS*. Base processor arithmetic instructions execute on the datapath portion shown with a generic register file, an ALU, two operand buses and one result bus. Custom instructions *MULT* and *MAC* perform their respective functionality (multiply and multiply-accumulate) on data values off the operand buses using shared custom hardware - a multiplier ($\times$), a multiplexer (*MUX1*), and an adder ($+1$). The multiplexer (*MUX1*) is used to select whether an immediate 0 (for *MULT* operation) or a real addend from custom register *ACCU* (for *MAC* operation) is fed to adder $+1$. Custom instruction *CUS* accesses custom registers *CR0*, $\dots$, *CR2* for its operands, which is independent of base processor operand buses. The computation results, either from base processor ALU or custom hardware modules, are fed into a multiplexer (*MUX2*). The result of execution is then selected and loaded onto the result bus, and stored into either generic registers or custom registers.

The description of the three custom instructions is shown in Fig. 2. In this figure, custom instruction keywords are shown in bold, and predefined names for the base processor core are underlined. The **user_register** statement specifies the custom state registers and their indices. The **iclass** statement defines a new instruction class with one or multiple custom

instructions. The input and output of this instruction class are also specified. The **semantic** statement describes the behavior of the instruction class in a single block. For some complex instructions which require multiple cycles, the **schedule** block gives the schedule for the operation sequence of the custom instruction [26], [27]. As seen from Fig. 2, state registers *ACCU* and *CR2* are both input and output for custom instructions. *MAC* and *MULT* instructions are in one *iclass*, and both take two cycles. The instruction timing in the schedule block shows that the *MAC* and *MULT* operations need *ars* and *art* at the beginning of the first cycle, use the *ACCU* value at the beginning of the second cycle and produce a new *ACCU* value at the end of the second cycle. The TIE compiler automatically derives the hardware implementation of custom instructions during the custom processor generation phase. It also offers dynamic linked libraries for instruction set simulation [27].

A snapshot of the dynamic execution of an application is captured in Fig. 1(b). Four instructions $I_1, \dots, I_4$ are shown in the trace, which correspond to base processor instruction *add* and custom instructions *MULT*, *MAC* and *CUS*, respectively. For each instruction, the top horizontal bar lists the sequence of processor events dictated by its execution. For example, instruction I_1 , which is an add instruction, executes by first reading data off the generic register file onto the operand buses (*RdReg*), then performing the add operation on the ALU and writing onto the result bus (*ALU*, *MUX2*), and writing back to the register file (*WrReg*) after a latency of one cycle due to pipeline effects (*Wait*). Stalls, if any, are also indicated in the figure.

Since the execution of an instruction can activate other portions of the processor (side effects), the bottom bar for each instruction depicts the side effects in either the base processor or the custom hardware. For example, the execution of the base processor instruction *add* activates custom hardware ($\times$, *MUX1*, $+1$) in the second cycle. This occurs because the custom hardware and the ALU of the base processor share the same operand buses. On the other hand, the execution of custom instructions (I_2 and I_3) can activate the base processor hardware, and the ALU is also activated when these custom instructions are running. Since the execution of the other custom instruction (I_4) may be independent of the base processor hardware, there is no side effect in the datapath.

B. Macro-model Identification

We analyze Fig. 1(b) further for energy consumption component identification. Along the horizontal time axis, processor events in boxes are time-disjoint since they occur in different cycles [21]. At the second stage of the pipeline (execution stage), as shown in the figure, the top box and bottom box are space-disjoint, which means that their functionalities involve different structural blocks [21]. For example, when base processor instruction I_1 is running, the top box represents base processor datapath events executed on *ALU* and *MUX2*, and the bottom box represents the possible side effect on custom hardware $+1$, *MUX1*, and $\times$; while if custom instruction I_2 is running, the top box shows the custom hardware activated,

(a)

(b)

Fig. 1. (a) Part of an extended processor, and (b) a snapshot of dynamic execution of a program on the processor

```

state CRO 32
state CR1 32
state CR2 32
user_register 0 CRO
user_register 1 CR1
user_register 2 CR2
iclass cc {CUS} {inout CR2, in CRO, in CR1}
semantic CUS {
    wire [31:0] temp1 = CRO + CR1;
    wire [31:0] temp2 = CR2 << temp1;
    assign CR2 = temp2;
}

state ACCU 32
user_register 3 ACCU
iclass cs {MAC, MULT} {inout ACCU, in ars, in art}
semantic {MAC, MULT}{
    wire [31:0] temp1 = ars × art;
    wire [31:0] temp2 = MAC ? ACCU : 0;
    assign ACCU = temp1 + temp2;
}
schedule {MAC, MULT}{
    use ars 1; use art 1;
    use ACCU 2; def ACCU 2;
}

```

Fig. 2. An example containing three custom instructions

i.e., $\times$, $+1$, $MUX1$, and $MUX2$, and the bottom box shows spurious activation of base processor ALU . Thus, since the activities in each box are either time-disjoint or space-disjoint to each other, their energy consumptions are additive.

To compute the energy consumption of the instruction trace in Fig. 1(b), we identify several macro-model components, and build an energy macro-model for extensible processors which accounts for the following factors.

- 1) The energy consumed by base processor instructions on the base processor core, for example, the top horizontal

bar for instruction I_1 execution in Fig. 1(b), which occurs throughout the five pipeline stages.

- 2) The energy consumed by a custom instruction on the custom hardware, for example, the second box in the top bar of I_2, I_3, I_4 , which is just computation energy.
- 3) The energy dependency on inter-instruction correlation, pipeline effects and other non-ideal features which are manifested as processor stalls, cache misses, *etc.*
- 4) The interplay between the base processor and custom hardware:

- On one hand, it contains the activation energy of custom hardware owing to base processor instructions, which is purely a computation side-effect (the bottom bar of instruction I_1 execution sequence in Fig. 1(b));
- On the other hand, it contains the activation energy of base processor hardware owing to custom instructions, which consists of both the computation side effect in the execution stage (the bottom bar of instructions I_2 and I_3 in Fig. 1(b)), and the involvement of the base processor in other pipeline stages (*RdReg*, *Wait*, *WrReg*, *WrCR* events in the top bar of instructions I_2 , I_3 , I_4).

Our proposed energy macro-model aims at gathering the most significant properties of extensible processors to define an accurate model efficiently. Since an extensible processor is augmented by a designer through custom instructions, the previous energy estimation work for microprocessors with fixed ISA is not applicable. The adopted macro-model is based on a decomposition of the activities carried out by the custom processor. Although the custom hardware is integrated with the base processor seamlessly, we can still decompose energy consumption into two space-disjoint parts: base processor and custom hardware extensions. We take a hybrid approach which utilizes both an instruction-level macro-modeling for the base processor and a structural approach for the custom hardware extensions.

IV. MACRO-MODELING AND ESTIMATION METHODOLOGY

In this section, we present the proposed methodology for estimating the energy consumption of an application with (any) custom instruction enhancements running on an extensible processor obtained for the application. Section IV-A presents an overview of our methodology, while Section IV-B details the constituent steps.

A. Overview

Fig. 3 shows the different steps involved in performing energy estimation for an extensible processor. The basic flow involves (a) characterization flow, which characterizes the energy consumption of the extensible processor through regression macro-modeling on a set of test programs (steps 1-8), and (b) estimation flow, which profiles a real application with custom instruction extensions dynamically to determine the statistics associated with the macro-model variables, and thus, computes the energy consumption (steps 9-11).

Building an energy macro-model involves first selecting the extensible processor parameters on which the energy consumption of an instruction trace depends and constructing a template that expresses the energy consumption (dependent variable) as a function of those parameters (independent variables) (step 1). The energy consumption is additive, and the processor events for the parameters in the macro-model are either time-disjoint or space-disjoint to each other, and hence there is no duplicate energy accounting during macro-modeling. We use a linear macro-model template in our analysis, since construction and

use of linear macro-models is efficient. The linear macro-model template¹ expresses the energy consumed, E , as a linear function of $X_1, X_2, \dots, X_n$, which are characteristic variables of a program running on the extensible processor. In other words,

$$E = E_0 + E_1X_1 + E_2X_2 + \dots + E_nX_n \quad (1)$$

where $E_0, E_1, \dots, E_n$ are constants called the energy coefficients. Variables $X_1, X_2, \dots, X_n$ are chosen from both instruction-level and structural domains. Instruction-level parameters are employed for characterizing instruction effects on the fixed base processor core, while structural parameters are used to characterize instruction effects on custom hardware due to any base or custom instructions (see Section IV-B).

The energy coefficients in the macro-model are determined using regression analysis (step 8). Since any test program can be used for building the regression model, we merely ensure that the different instructions in the base processor ISA are covered (step 2). The input space of custom instructions that can be added to an extensible processor ISA is, however, exponential for a given choice of custom hardware library components. Therefore, the test program suite also incorporates custom instructions so as to cover all the custom hardware library components (step 2). Regression analysis requires knowledge of both the dependent variables (energy consumption for test programs) and the independent variables (macro-model parameters values). For each test program, instruction set simulation (step 6) and dynamic resource usage analysis (step 7) on the execution trace are used to determine the values of the parameters in the macro-model template. Energy estimation of the test program executing on the RTL description of the processor (step 5) is used to measure the value of the dependent variable. If the test program incorporates custom instructions, the processor used in the above step corresponds to a custom processor that includes the custom instructions (through step 4). Note that, while custom processors are generated during characterization, they are not needed for using the macro-model to compute application energy consumption. The macro-modeling is just a one-time effort. Steps 3-7 are repeated for all the test programs to gather the data needed by regression macro-modeling to find estimates of the energy co-efficients, thus completing characterization.

When the energy consumption of an application with custom instructions has to be estimated (step 9-11), instruction set simulation (step 9) is first performed to gather execution statistics (values of instruction-level macro-model variables) as well as to generate the dynamic execution trace for value extraction of structural variables. Since the integration of custom hardware (due to the custom instructions) with the processor architecture is defined in an extensible processor design flow, we analyze the resource usage (step 10) of each instruction in the execution trace to determine the activation (if any) of custom hardware. This analysis yields the values of

¹Note that linear macro-models require linearity in the coefficients, not in the parameters. Hence, polynomial and other non-linear functions of macro-model parameters can be present in the template.

Fig. 3. Extensible processor energy estimation flowchart

the structural macro-model variables. The parameter values are fed to the energy macro-model to yield the energy consumed by the application.

B. Details

In this section, we focus on the implementation of salient steps of our methodology. Section IV-B.1 discusses energy macro-model template generation, while Section IV-B.2 examines macro-model fitting using regression analysis.

1) *Energy Macro-model Template Generation*: The energy macro-model template used in our analysis is linear, with two components as shown below.

$$E = E_{ins}(X_1, \dots, X_m) + E_{struc}(X_{m+1}, \dots, X_n) \quad (2)$$

where E_{ins} and E_{struc} are linear functions of instruction-level parameters $(X_1, \dots, X_m)$ and structural parameters $(X_{m+1}, \dots, X_n)$, respectively. They are space-disjoint, as E_{ins} depicts energy consumption on the base processor due to

program execution, and E_{struc} on custom hardware due to all the instructions in program.

Instruction-level Macro-model Variables: The instruction-level macro-model variables are chosen to reflect the usage of the base processor core due to either base processor or custom instructions. In the process, we also select parameters to consider the effects of non-ideal processor cases caused both by a base processor instruction and custom instruction, such as instruction cache miss, data cache miss, uncached instruction fetching, data or control dependent interlocks, *etc.*, that occur during the execution of a program.

Equation (3) illustrates the use of instruction-level macro-model variables to compute E_{ins} .

$$\begin{aligned} E_{ins} = & E_{arith} \cdot Cyc_{arith} + E_{ld} \cdot Cyc_{ld} + E_{st} \cdot Cyc_{st} + \\ & E_j \cdot Cyc_j + E_{br_tk} \cdot Cyc_{br_tk} + E_{br_utk} \cdot Cyc_{br_utk} + \\ & E_i \cdot Num_i + E_d \cdot Num_d + E_{uncache} \cdot Num_{uncache} + \\ & E_{interlock} \cdot Num_{interlock} + E_{side_tie} \cdot Cyc_{side_tie} \end{aligned} \quad (3)$$

wherein E_{ins} is expressed as a linear sum of the following

energy components:

- Energy due to the base processor functionality exercised by an instruction belonging to the base processor ISA (energy component item (1) in Section III-B). Experimental studies of energy profiles of processor instructions in the literature suggest that instructions in the base processor ISA can be clustered into arithmetic (*arith*), load (*ld*), store (*st*), jump (*j*), branch taken (*br_tk*), and branch untaken (*br_unk*) classes [15]. We also ran a set of test programs for each base ISA instruction and classified them into these categories according to their average energy consumption. Macro-model variables $Cyc_{arith}, \dots, Cyc_{br_unk}$ represent the number of cycles taken by each instruction class in the dynamic execution trace of the program. Such a clustering is convenient (and later seen to be accurate) since macro-model variables do not have to be separately present for individual instructions in the base processor ISA. Thus, efficiency is targeted by instruction clustering.
- Energy due to dynamic effects manifested as instruction-cache misses (*i*), data-cache misses (*d*), uncached instruction fetches (*uncache*) and processor interlocks (*interlock*) [15] (item (3) in Section III-B). Macro-model variables $Num_i, \dots, Num_{interlock}$ denote the number of times each non-ideal case occurs during program execution.
- Energy consumption in the base processor imposed by custom instructions (*side_tie*). This corresponds to the second energy component of item (4) in Section III-B. It actually contains two components, one is for all the custom instructions, which is the energy consumption in the four pipeline stages other than the execution stage; the other one is for those custom instructions that access the generic register file, which is the spurious energy consumption in the ALU activated in the execution stage. Since the latter computation energy is very small, we only account for the former one. The macro-model variable Cyc_{side_tie} accounts for the number of cycles taken by all the custom instructions.

Structural Macro-model Variables: Structural macro-model variables reflect the usage of custom hardware extensions due to the execution of either the base processor or custom instructions. The variables are chosen to account for the number of cycles for which each custom hardware component is active during the execution of a program. All the components present in the custom hardware library should, therefore, be covered by these macro-model variables. The custom hardware extensions can be designed to be large enough to match or even exceed the size of the base processor core according to the requirement of specific applications. For example, the present generation of Xtensa III cores range in complexity from a base configuration with approximately 25,000 gates to configurations with several hundred thousand gates [6]. Hence, accurate modeling of custom hardware plays an important role in macro-modeling of the whole extended processor. The following considerations are made in the process.

Custom Hardware Module Categorization: For efficiency,

the components in the custom hardware library are classified into several categories based on empirical studies of their average energy consumption, even though their functionalities differ from each other. In the context of the custom hardware library used by TIE instructions, we classified the basic primitives into five categories: (1) multiplier ($\times$), (2) adder, subtractor and comparators ($+$, $-$, $>$, $<$, $\dots$), (3) bit-wise logic ($\wedge$, $\&$, $|$, $\dots$), reduction logic ($\sim\wedge$, $\sim\&$, $\sim|$, $\dots$), and multiplexers ($? :$), (4) shifter ($<<$, $>>$), and (5) custom registers. Additional categories correspond to specialized modules available for TIE instructions, namely, custom built-in modules: (6) TIE_mult, (7) TIE_mac, (8) TIE_add, and (9) TIE_csa, and (10) table, which is a hardware implementation that stores constants to be used in computational expressions.

Energy Complexity Scaling for Hardware Modules: The energy consumption of a hardware component depends significantly on its bit-width (or the number of entries and bit-width of each entry in the case of a *table*). We use C to represent the power complexity of a hardware module and n_b for the bit-width. The dependence on bit-width is linear ($C \propto n_b$) in the case of hardware components such as ripple-carry adders, multiplexers, *etc.*, while the dependence is quadratic in the case of an array multiplier ($C \propto n_b^2$). The linear categories include (2), (3), (4), (5), (8), (9), and the quadratic categories include (1), (6), (7). Energy consumption of the static *table* is linearly dependent on both the bit-width and number of entries in *table*. We expect accuracy to be gained through this energy complexity scaling for custom hardware modules.

Dynamic Resource Usage Analysis: Note that since for each custom hardware module, there are several factors that decide whether it is in an active state or idle state, we need to dynamically analyze the resource usage pattern, which is referred to in steps 7 and 10 in Fig. 3. A custom functional block is activated when any custom instruction belonging to the corresponding custom instruction class is executing. For example, the shared hardware modules ($\times$, *MUX1*, $+$) in Fig. 1(a) are activated when either the *MULT* or *MAC* instruction is executing. In some cases, the custom functional block can also be activated when base processor instructions are running. If the functional block has inputs from generic registers, or it is in the datapath chain of generic register value propagation, the activities on the operand buses still affect the custom hardware. Hence, we also account for the usage of processor extensions due to base processor instructions. The execution traces obtained from instruction set simulation (step 6 in Fig. 3) provide us the information on sequential instruction execution. Since the TIE compiler just implements the custom instructions automatically by mapping each operation to the corresponding operator one-by-one, we employ a data-flow-graph (DFG) for each custom instruction, with nodes (functional blocks) corresponding to operations, and edges corresponding to data flow, as shown in the custom hardware extensions in Fig. 1(a). Thus, we identify the activated custom hardware modules for each instruction in the execution trace, and count the activation cycles for each module.

Equation (4) expresses the custom hardware energy consumption, E_{struc} , where macro-model variables $Cyc_{i,j}$ ($i = 1, 2, \dots, 10$) denote the number of cycles in which the j th

functional block belonging to component category i is active, and $C_{i,j}$ represents the energy complexity of this functional block. E_i ($i = 1, 2, \dots, 10$) represents the average energy consumption per bit (per entry for *table*) per cycle for each kind of resource module category.

$$E_{struct} = E_1 \cdot \sum_j C_{1,j} C y c_{1,j} + E_2 \cdot \sum_j C_{2,j} C y c_{2,j} + \dots + E_{10} \cdot \sum_j C_{10,j} C y c_{10,j} \quad (4)$$

2) *Macro-model Fitting through Regression Analysis*: Regression analysis is used to determine the energy coefficients in the macro-model template shown in Equation (2) (with E_{ins} and E_{struct} as given in Equations (3) and (4), respectively). Test programs are used to gather both energy consumption values and execution statistics that correspond to the different macro-model variables in the equation. For a set of n test programs, the energy consumption data are grouped into an $n \times 1$ column vector, E , while the values corresponding to the macro-model variables are grouped into an $n \times 21$ matrix, M . In such a case, model fitting through regression analysis involves solving the linear matrix equation, $M \times C = E$, where C is the energy coefficient vector corresponding to $\{E_{arith}, E_{ld}, E_{st}, E_j, E_{br_tk}, E_{br_utk}, E_i, E_d, E_{uncache}, E_{interlock}, E_{side_tie}, E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8, E_9, E_{10}\}$.

In the above formulation, let $\tilde{C}$ represent the estimate of C , and $\tilde{E}$ represent the estimate of E . A solution of the matrix equation using the pseudo-inverse method [28] yields the values for the energy coefficient vector C , as shown in Equation (5), such that the mean square error, as shown in Equation (6), is minimized.

$$\tilde{C} = (M^T \times M)^{-1} \times M^T \times E \quad (5)$$

$$\|E - \tilde{E}\| = \sqrt{\frac{\sum_{i=1}^n ((E_i - \tilde{E}_i)/E_i)^2}{n}} \quad (6)$$

The macro-model with the above energy coefficient values best fits the data acquired using the test programs. To estimate the model coefficients $\tilde{C}$, the rows of matrix M must be linearly independent, otherwise the problem has infinitely many solutions and the macro-model is *not identifiable* to energy consumption values of the test programs [21]. Because matrix $M^T \times M$ is singular in this case, i.e., $|M^T \times M| = 0$, the pseudo-inverse $M^* = (M^T \times M)^{-1} \times M^T$ cannot be calculated. To avoid the linear dependency of two rows in matrix M , we should induce enough diversity in the test programs, so that they differ with respect to functionalities of macro-model parameters, and the number of test programs should be no smaller than the number of model parameters.

V. EXPERIMENTAL RESULTS

We have implemented the proposed flow described in Section IV using commercial tools and scripts to perform several key tasks in the methodology. The target extensible processor used in our experiments is the Xtensa processor from Tensilica [1]. The base processor is a T1040.0 version of the

processor running at 187 MHz (0.18 μ technology), and the configuration includes a 32-bit multiplication instruction, 4-way 16 KB instruction and data caches, 32-bit wide system bus, and a generic register file with 64 32-bit registers. The processor includes clock-gating features.

Characterization in our set-up proceeds as follows. The GNU-based cross-compiler of the Xtensa software development kit is used to cross-compile test programs to get their memory images (step 3 in Fig. 3). An instruction set simulator is used to simulate test programs to gather execution statistics (cycle count, cache misses, interlock, *etc.*) needed by the macro-model (steps 6, 9). Test programs are Tensilica benchmarks written in C, while custom instructions are written in the TIE language [1]. Test programs instantiate TIE instructions intrinsic to their description, which are cross-compiled to generate executables for instruction set simulation. The Xtensa processor generator is used to generate the RTL description of the custom processors needed during characterization. The RTL description is simulated with the memory images of the test programs using ModelSim [29] to generate the simulation traces needed by the RTL power estimator WattWatcher [13] (steps 5). The NEC RTL cell library CB11 for the 0.18 μ m technology [30] is used for WattWatcher. The energy values and execution statistics for the set of test programs thus obtained are used by the tool Splus [31] to perform regression and derive the energy macro-model. The results of characterization are presented in Section V-A, while the results of applying the macro-model to evaluate the energy consumption of different applications are presented in Section V-B.

A. Energy Macro-model for the Xtensa Processor

The energy macro-model template for the Xtensa processor is shown in Equation (2), with the instruction-level and structural macro-model terms given in Equations (3) and (4), respectively. There are 21 macro-model variables in the template, whose coefficients are determined through regression analysis. Table I presents these coefficients which indicate the per-cycle estimates of the base processor energy consumption for each base processor instruction category ($E_{arith}, \dots, E_{br_utk}$), the per-cycle estimates of the energy consumption of the side-effects of custom instructions on the base processor (E_{side_tie}), and the per-miss/per-fetch/per-interlock estimates of the energy consumptions for execution-time effects ($E_i, \dots, E_{interlock}$), as well as the unit energy consumption (per-cycle, per-bit) for the different custom hardware library components ($E_1, \dots, E_{10}$).

All the test programs used for characterization are benchmarks used by Tensilica to evaluate the performance of their Xtensa-series embedded processors. The test program suite includes programs from a wide range of applications to (a) cover the various instructions in the instruction set, (b) exercise the various architectural manifestations such as instruction and data cache misses, processor interlocks, *etc.*, and (c) cover scenarios so that all the custom hardware modules are used. The test programs include applications such as *rgb2cmypk*, *alphablend*, *color* from image processing and graphics, *solomon*, *rand* from wireless communication, *checksum*, *byteswap* from

TABLE I
ENERGY COEFFICIENTS OF THE CHARACTERIZED XTENSA PROCESSOR

Energy coefficient	Description	Value
E_{arith}	arithmetic instruction	$0.638nJ$
E_{ld}	load instruction	$0.512nJ$
E_{st}	store instruction	$1.092nJ$
E_j	jump instruction	$0.603nJ$
E_{br_tk}	branch taken	$0.504nJ$
E_{br_utk}	branch untaken	$0.345nJ$
E_{side_tie}	side effects due to custom instructions	$0.616nJ$
E_i	instruction cache miss	$2.93nJ$
E_d	data cache miss	$4.34nJ$
$E_{uncache}$	uncached instruction fetch	$2.42nJ$
$E_{interlock}$	processor interlock	$0.98nJ$
E_1	*	$152.0 fJ$
E_2	+/-/comp	$70.0 fJ$
E_3	log/red/mux	$12.0 fJ$
E_4	shifter	$377.0 fJ$
E_5	custom register	$177.0 fJ$
E_6	TIE_mult	$165.0 fJ$
E_7	TIE_mac	$190.0 fJ$
E_8	TIE_add	$69.0 fJ$
E_9	TIE_csa	$37.0 fJ$
E_{10}	table	$27.0 fJ$

networking, *audio* from voice compression, *add4*, *bubsort*, *gcd* from digital signal processing, *solomon* from network security, *etc.* Note that any test program suite can be used for characterization as long as the above-mentioned requirements are met, and the set of programs is diverse enough not to bias characterization.

Fig. 4 reports the fitting errors corresponding to each test program. The errors are very small with the maximum error being 8.9%, and the root mean square error being 3.8%. We observed that the fitting errors of the test programs *solomon* and *color* are higher than the errors of the other test programs. Note that the overall macro-model is built using a *linear macro-model template* based on the statistics corresponding to *all* the test programs. Therefore, it is possible for regression analysis to be unable to fit some of the statistics corresponding to *solomon* and *color* to a linear macro-model template, while attempting to minimize the fitting error corresponding to *all* the test programs. However, the overall accuracy of the macro-model will be evaluated by using it for any test application as shown in the next section.

B. Applying the Energy Macro-model: Accuracy Results

In this section, we present the results of applying the energy macro-model described in Section V-A. We performed two experiments to validate our methodology and the macro-model.

1) *Absolute Accuracy Examination:* Our first experiment involves determining the energy consumption of several arbitrary applications (incorporating different custom instructions) in two ways: (i) using the derived energy macro-model, and (ii) using the RTL power estimation tool WattWatcher [13] on the synthesized RTL description of the corresponding

extended processor. Table II summarizes these results. For the ten application benchmarks (different from the test programs used in characterization flow) shown in the first column, the maximum estimation error is 8.5%, while the average absolute error is only 3.3%. The description for each application is given in the second column.

Fig. 5 shows the breakdown of energy consumption of some applications into two parts: base processor core and custom hardware extension. For each application, the left bar represents the energy estimation by our macro-model for both the base processor core and custom hardware extensions, and the right bar represents the corresponding simulation results obtained from WattWatcher. The results show that (i) the energy consumption due to custom hardware can be significant, for example, for application *Multi_accumulate*, the energy consumption ratio of custom hardware extension and the base processor core is 34.0% (the average ratio for these applications is 27.3%), and (ii) the performance of the macro-model is good both for the base processor and custom hardware, as seen from the graph, our macro-model tracks the results reported by WattWatcher in both the cases.

The proposed energy estimation methodology is very fast. It takes only a few seconds for application energy estimation using our approach, and it does not need RTL generation of the custom processor. The average time taken by WattWatcher to determine the energy consumption of a small application is several hours, and it needs the RTL description of the custom processor, which takes several more hours to generate. Thus, we achieve an average speedup of three orders of magnitude.

2) *Relative Accuracy Examination:* We also performed an additional experiment to study the relative accuracy of the macro-model, when used in energy optimization studies for an

Fig. 4. Fitting error of the test programs for extensible processor cores

TABLE II
APPLICATION ENERGY ESTIMATES: ACCURACY RESULTS

Application	Description	Estimate (μJ)	WattWatcher (μJ)	Error (%)
Ins_sort	insertion sort of an array of specified size	336.9	344.5	-2.2
Gcd_2	calculate the greatest common divisor of two numbers	736.5	723.5	1.8
Alphablend_3	alphablend two 24-bit pixels	106.9	105.7	1.1
Add4_3	add four 16-bit values stored in two operands	595.0	583.9	1.9
Bubsort	bubble sorting algorithm	131.5	126.7	3.8
DES	DES encryption algorithm	45.6	43.7	4.3
Accumulate	implement a 40-bit accumulator	37.6	35.4	6.2
Drawline	accelerate the line draw process by Bresenham's algorithm	9.9	9.7	2.0
Multi_accumulate	implement a 64-bit MAC	23.8	26.0	-8.5
Seq_mult	implement a 16-bit sequential circuit binary multiplier	13.5	13.7	-1.5

application with multiple custom instruction choices (note that each choice may contain a group of custom instructions). For this experiment, we implemented the Reed-Solomon decoding/encoding algorithm with four different custom instruction choices. Fig. 6 shows the energy estimates obtained by using our macro-model for each choice (this required merely ISS simulation to generate the statistics required by our macro-model). For each choice, we also obtained the energy estimates using WattWatcher (this required the actual synthesis of the custom processor in each case, followed by RTL simulation and power estimation using WattWatcher). The two profiles track each other, demonstrating good relative accuracy of our macro-model. The high relative accuracy and low effort characteristics (no custom processor generation, no RTL simulation) associated with the usage of our macro-model makes our approach highly suitable for such architectural exploration studies.

VI. CONCLUSIONS

In this work, we presented an efficient framework for characterizing the energy consumption of an extensible processor.

Our characterization flow facilitates the construction of an efficient macro-model by using parameters drawn from both instruction-level and structural domains, and by leveraging the benefits of regression analysis. Since application of the macro-model only requires instruction set simulation based analysis of the application, energy estimation using our approach is very fast. We characterized the energy consumption of a state-of-the-art extensible processor and used the macro-model so obtained to derive highly accurate energy estimates for several applications.

Since our goal was to estimate the energy consumption of an application running on an extensible processor efficiently and accurately, we made some conscious choices while developing our energy macro-model characterization and usage flows. High efficiency comes from choosing parameters in the macro-model template, which are available from instruction-level simulation. Some of the instruction-level parameters account for the dynamic characteristics of a processor (such as processor stalls) and memory hierarchy (such as cache misses and uncached instruction fetch overheads). Moreover, we performed a dynamic analysis of custom hardware usage

Fig. 5. Energy consumption breakdown between the base processor and custom hardware extension derived using our macro-model and WattWatcher

Fig. 6. Energy consumption estimates of an application for various custom instruction choices derived using our macro-model and WattWatcher

pattern with the instruction execution trace, which captures the computational functionality of custom processor extensions, thereby, improving the overall accuracy. Further accuracy improvements can be made by considering explicitly other inter-instruction effects and data value dependencies. In addition, some processors may require consideration of effects such as deeper pipeline stages, longer instruction word, instruction encoding, addressing mode, activity on data bus, *etc.* for better accuracy (with possibly slower estimation times). We believe that the proposed energy estimation technique provides a fundamental framework that can be extended to tackle the characterization requirements of other complex processors.

REFERENCES

- [1] Xtensa microprocessor, tensilica Inc. (<http://www.tensilica.com>).
- [2] ARCtangent processor, arc International (<http://www.arc.com>).
- [3] Jazz DSP, improv Systems Inc. (<http://www.improvsys.com>).
- [4] SP-5flex DSP core, 3DSP Corp. (<http://www.3dsp.com>).
- [5] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Synthesis of custom processors based on extensible platforms," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2002, pp. 641–648.
- [6] J. Sanghavi and A. Wang, "Estimation of speed, area, and power of parameterizable soft IP," in *Proc. Design Automation Conf.*, June 2001, pp. 31–34.
- [7] J. Rabaey and M. Pedram (Editors), *Low Power Design Methodologies*. Kluwer Academic Publishers, Norwell, MA, 1996.
- [8] A. Raghunathan, N. K. Jha, and S. Dey, *High-level Power Analysis and Optimization*. Kluwer Academic Publishers, Norwell, MA, 1998.
- [9] P. W. Ong and R. H. Yan, "Power-conscious software design - A framework for modeling software on hardware," in *Proc. Int. Symp. Low Power Electronics & Design*, Oct. 1994, pp. 36–37.
- [10] H. Mehta, R. M. Owens, and M. J. Irwin, "Instruction level power profiling," in *Proc. Int. Conf. Acoustics Speech & Signal Processing*, May 1996, pp. 3326–3329.
- [11] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. Int. Symp. Computer Architecture*, June 2000, pp. 83–94.
- [12] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of SimplePower: A cycle-accurate energy estimation tool," in *Proc. Design Automation Conf.*, June 2000, pp. 340–345.
- [13] *WattWatcher manual*, sente Inc. (<http://www.sequencedesign.com>).
- [14] C.-T. Hsieh, M. Pedram, G. Mehta, and F. Rastgar, "Profile-driven program synthesis for evaluation of system power dissipation," in *Proc. Design Automation Conf.*, June 1997, pp. 576–581.
- [15] 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.
- [16] J. Russell and M. Jacome, "Software power estimation and optimization for high-performance 32-bit embedded processors," in *Proc. Int. Conf. Computer Design*, Oct. 1998, pp. 328–333.
- [17] N. Chang, K. Kim, and H. G. Lee, "Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 2000, pp. 185–190.
- [18] A. Sama, M. Balakrishnan, and J. F. M. Theeuwens, "Speeding up power estimation of embedded software," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 2000, pp. 191–196.
- [19] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proc. Int. Wkshp. Power & Timing Modeling, Optimization & Simulation (PATMOS)*, Sept. 2001.
- [20] G. Ascia, V. Catania, M. Palesi, and D. Sarta, "An instruction-level power analysis model with data dependency," *VLSI Design, Special Issue on Low Power System Design*, vol. 12, pp. 245–273, 2001.
- [21] C. Brandolese, F. Salice, W. Fornaciari, and D. Sciuto, "Static power modeling of 32-bit microprocessors," *IEEE Trans. Computer-Aided Design*, vol. 21, no. 11, pp. 1306–1316, Nov. 2002.
- [22] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "Instruction-level power estimation for embedded VLIW cores," in *Proc. Int. Wkshp. Hardware/Software Codesign*, Mar. 2000, pp. 34–38.
- [23] S. Lee, A. Ermedahl, S. L. Min, and N. Chang, "An accurate instruction-level energy consumption model for embedded RISC processors," in *Proc. ACM SIGPLAN Wkshp. Languages, Compilers, & Tools for Embedded Systems*, June 2001, pp. 1–10.
- [24] C. H. Gebotys, R. J. Gebotys, and S. Wiratunga, "Power minimization derived from architectural-usage of VLIW processors," in *Proc. Design Automation Conf.*, June 2000, pp. 308–311.
- [25] T. Kimura, S. Shibahara, Y. Takeuchi, M. Imai, A. Kitajima, and M. Muraoka, "A power estimation method for ASIPs considering data types of variables in application programs," in *Proc. Int. Tech. Conf. Circuits/Systems, Computers & Communications*, July 2002, pp. 387–390.
- [26] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, Mar.-Apr. 2000.
- [27] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-chip processors," in *Proc. Design Automation Conf.*, June 2001, pp. 184–188.
- [28] S. Chatterjee, A. S. Hadi, and B. Price, *Regression Analysis by Examples, 3rd ed.* John Wiley & Sons, 2000.
- [29] *ModelSim*, model Technology (<http://www.model.com>).
- [30] *CB-11 Cell Based IC Product Family*, NEC Electronics, Inc. (<http://www.necelam.com/asic/cell-based.cfm>).
- [31] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S-PLUS*. Springer-Verlag, 1998.