2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Systematic Energy Characterization of CMP/SMT Processor Systems via Automated Micro-Benchmarks

Ramon Bertran[*†] Alper Buyuktosunoglu[†] Meeta S. Gupta[†] Marc Gonzàlez[*] Pradip Bose[†]

[*]Barcelona Supercomputing Center
C. Jordi Girona 29, Barcelona, Spain
{ramon.bertran,marc.gonzalez}@bsc.es

[†]IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
{rbertra,alperb,mgupta,pbose}@us.ibm.com

Abstract

Microprocessor-based systems today are composed of multi-core, multi-threaded processors with complex cache hierarchies and gigabytes of main memory. Accurate characterization of such a system, through predictive pre-silicon modeling and/or diagnostic post-silicon measurement based analysis are increasingly cumbersome and error prone. This is especially true of energy-related characterization studies. In this paper, we take the position that automated micro-benchmarks generated with particular objectives in mind hold the key to obtaining accurate energy-related characterization. As such, we first present a flexible micro-benchmark generation framework (MicroProbe) that is used to probe complex multi-core/multi-threaded systems with a variety and range of energy-related queries in mind. We then present experimental results centered around an IBM POWER7 CMP/SMT system to demonstrate how the systematically generated micro-benchmarks can be used to answer three specific queries: (a) How to project application-specific (and if needed, phase-specific) power consumption with component-wise breakdowns? (b) How to measure energy-per-instruction (EPI) values for the target machine? (c) How to bound the worst-case (maximum) power consumption in order to determine safe, but practical (i.e. affordable) packaging or cooling solutions? The solution approaches to the above problems are all new. Hardware measurement based analysis shows superior power projection accuracy (with error margins of less than 2.3% across SPEC CPU2006) as well as max-power stressing capability (with 10.7% increase in processor power over the very worst-case power seen during the execution of SPEC CPU2006 applications).

1. Introduction

The power wall has proven to be a major obstacle in the quest to sustain the historical rates of performance growth in computing systems. The multi-core/multi-threaded design paradigm (CMP/SMT) has enabled the growth of throughput performance despite the dramatic slowdown in clock speed growth. However, power dissipation and current delivery limits make it hard to keep scaling indefinitely along the dimension of on-chip thread count. As such, it is important to understand the limits and sensitivities of energy-related metrics associated with current generation processors —so that future systems can invest into appropriate levels of power management in the right regions of the micro-architectural design space.

Microprocessor systems today are composed of multi-core, multi-threaded processors with complex cache hierarchies and gigabytes of memory. Predictive pre-silicon modeling and diagnostic post-silicon measurement studies are increasingly cumbersome and error prone. When it comes to power or energy-related metrics, the challenge is especially steep, since fine-grained power measurements or predictions across a complex, highly-threaded multi-core system are quite difficult. In this paper, we take the position that micro-benchmarks, generated with particular objectives in mind hold the key to obtaining accurate energy-related characterization. Specially crafted micro-benchmarks may be run on simulators (pre-silicon stage) or real machines (post-silicon stage) to help understand, diagnose and fix deficiencies systematically. However, manual generation of such 'stressmarks' is tedious, and requires intimate knowledge of the underlying micro-architecture pipeline semantics. Automated micro-benchmark generation is therefore crucial in this regard. Moreover, the automated generation facility must be flexible enough to generate different classes of micro-benchmarks that are useful in answering a range of different questions.

In this paper, we present a flexible micro-benchmark generation framework (MicroProbe) that is used to probe complex processor systems with a variety and range of energy-related queries in mind. In particular, three different characterization queries are illustrated in this paper. MicroProbe's automated generation facility is used to derive: (a) an accurate and decomposable power model that is used to project the power consumption for arbitrary CMP/SMT workloads; (b) energy-per-instruction (EPI) ratings for different instruction classes supported by the system; and (c) a systematically generated synthetic stress test that maximizes power consumption for the targeted system[1]. Experimental results are measured on a POWER7-based 8-core/32-thread system [42] in order to validate the efficacy of MicroProbe.

This is quite different from prior work [13,20,33,34,39,40], where 'black-box' automatic test case generators are focused on stressing or validating only a single metric: e.g. IPC, power or some utilization-based index of performance or power. MicroProbe presents the following unique features: *Detailed knowledge of low-level micro-architecture semantics* to assist the micro-benchmark generation process ('white-box' approach), a compiler-like *pass-based code generator* to provide flexibility and full control over the code being generated, and highly integrated *design space exploration support* to search for optimal micro-benchmarks. Overall, MicroProbe increases the productivity of the investigative micro-architect as he/she stresses the system to understand the fundamental trade-offs across power and performance metrics.

The main contributions of this paper are the following:

- We present the software architecture of *MicroProbe*: a framework for automated generation of micro-benchmarks that a user can adapt towards exercising a complex multi-core, multi-threaded computing system in a variety of redundant ways for answering a range of questions related to energy and performance. The illustrative use of MicroProbe in this paper is limited to three low-level energy-related case studies as stated below.
- We show how targeted micro-benchmarks generated by MicroProbe can be used to form a bottom-up power model that is able to predict general CMP/SMT workload power very accurately. To the best of our knowledge, this is the first bottom-up counter-based power model for a CMP/SMT processor. This type of models are

[1]From now on, we refer to it as *max-power stressmark*. This type of test cases are also known *synthetic TDP workloads*.

known to perform better [7–9] than those derived from common modeling approaches. However, they had limited applicability due to the lack of frameworks for automating the generation of the micro-architecture aware training data that they need. We show, through real measurements, that the model is able to predict POWER7 processor power consumption with average error of only 2.3% across the SPEC CPU2006 benchmarks. We compare the model against a set of models generated using existing approaches to show that the generated model outperforms existing approaches, even on extreme power situations. Finally, we use the extra information provided by the model to present the average SPEC CPU2006 processor power breakdown for different POWER7 SMT/CMP modes.

- We develop a taxonomy of the POWER7 instructions based on energy per instruction (EPI) and processor activity characteristics. As far as we know, this is the first EPI-based taxonomy at instruction level for a CMP/SMT processor such as the POWER7. We report up to 78% variations on EPI values across instructions, even when they stress the same functional unit at the same rate. These findings highlight the importance of such taxonomies in understanding the instruction-level power-performance trade-offs.
- We use EPI and IPC based formalisms to generate max-power synthetic stress test programs using the MicroProbe facility. Prior methods [20, 21, 33, 40] use abstract workload models to make the design space tractable, losing therefore opportunities during the instruction type selection pass. We exploit the rich information implemented in MicroProbe to use the instructions with higher EPI and IPC per functional unit as the building blocks of the max-power stressmark. Exhaustive exploration performed on that small subset of selected instructions was able to find a stress test that exceeds the maximum power seen during the full-suite SPEC 2006 benchmark execution by 10.7%. We also report that stressmarks with the same instruction type distribution and activity rate but different instruction order can show up to 17% difference in power consumption. The fact that the systematically generated stressmark slightly outperformed the hand-crafted stress tests generated by an expert confirms the utility of the proposed approach.

The rest of the paper is organized as follows. Section 2 explains the software architecture of the micro-benchmark generation framework. The hardware evaluation and experimental measurement platform is described in Section 3. Sections 4, 5 and 6 discuss the case studies. Section 7 summarizes the related research work and Section 8 provides concluding remarks.

2. MicroProbe framework

An overview of the design of MicroProbe and its usage flowchart is shown in Figure 1. MicroProbe provides a Python scripting interface to access to a rich set of mechanisms and features. The interface allows the users to identify the architecture components and their parameters in order to accommodate the micro-benchmark design to a target architecture. We show some examples highlighting the variety of possible user-defined micro-benchmark generation policies above the dotted horizontal line in Figure 1.

In MicroProbe, the micro-benchmarks are represented by a specific internal representation within the *Code generation module*. This representation can be transformed by a sequence of passes driven by the micro-benchmark synthesizer. The micro-benchmark synthesizer is in charge of generating the final code by applying the passes ordered in accordance to user-specified ordering rules. MicroProbe

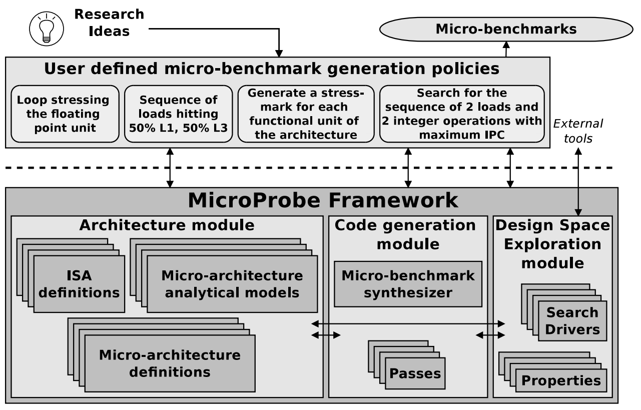

Research Ideas
Micro-benchmarks
User defined micro-benchmark generation policies
Loop stressing the floating point unit
Sequence of loads hitting 50% L1, 50% L3
Generate a stress-mark for each functional unit of the architecture
Search for the sequence of 2 loads and 2 integer operations with maximum IPC
External tools
MicroProbe Framework
Architecture module
ISA definitions
Micro-architecture analytical models
Micro-architecture definitions
Code generation module
Micro-benchmark synthesizer
Passes
Design Space Exploration module
Search Drivers
Properties

Figure 1: MicroProbe usage flowchart (top) and its design overview (bottom). The modular design provides flexibility in all the steps of the micro-benchmark generation process.

is therefore a framework that operates like a compiler infrastructure, achieving a high degree of flexibility and adaptability.

MicroProbe makes the whole micro-benchmark design process portable to different architectures. This feature is achieved by separating the architecture dependencies from the process itself (*Architecture module* in Figure 1). MicroProbe allows the user to describe the architecture through a set of readable text files where the architecture components and their parameters are set (the Instruction Set Architecture (ISA) and Micro-architecture definitions in Figure 1). All this information, in conjunction with micro-architecture analytical models, can be used (queried) to guide the micro-benchmark generation process. The generated micro-benchmarks are therefore bound to a specific architecture, but not the generation process.

Automated design space explorations (DSE) are required to assist the generation of micro-benchmarks with dynamic properties that cannot be ensured statically. MicroProbe integrates support for performing automatized DSEs within the *Design space exploration module* (See Figure 1). This module defines the mechanisms and features required to allow the user to define the design space and the search algorithm. Thus, MicroProbe is seen to provide full flexibility to perform any kind of DSE.

The modular design with standardized interfaces between the modules (as shown in Figure 1) makes the framework adaptable. In addition to this virtue, the following novel functionalities incorporated in MicroProbe advance the state of the art significantly:

MicroProbe is guided by low-level microarchitecture semantics. This is an important feature that was missing in previous work. This information is crucial to assist the generation of micro-architecture aware micro-benchmarks. It provides a 'white-box' solution to the users to define micro-benchmarks with very specific micro-architecture properties, avoiding the need to master every detail of the complex underlying architectures. We explain the Architecture module in detail in Section 2.1.

MicroProbe also presents novelty in the flexible code generation support (*Code generation module*) and in the *integrated* design space exploration (DSE) module. These functionalities, not available in previous work, improve the productivity and range of applicability of the micro-benchmark generation framework. In Section 2.2, we discuss the benefits of the compiler-like pass-based design of the *Micro-benchmark synthesizer* and in Section 2.3, we present the advantages of the integrated generic DSE support.

The rest of the section details the novel aspects within the three

```
1  import MicroProbe as MP
2  # Get the architecture object
3  arch = MP.arch.get_architecture("POWER7")
4  # Create the micro-benchmark synthesizer
5  synth = MP.code.Synthesizer(arch)
6  # Add the passes to be used
7  # to synthesize micro-benchmarks.
8  # Pass 1: Define the program skeleton
9  synth.add_pass("Single end-less loop
10                 of 4096 instructions")
11 # Pass 2: Define the instruction distribution
12 # Pass 2.1: Select the loads from the ISA
13 loads = [ Select ins in  arch.isa() if ins.load() ]
14 # Pass 2.2: Select the VSU Unit loads
15 loads_vsu = [ Select ins in loads
16               if  ins.stress(arch.comps["VSU"]) ]
17 synth.add_pass("Distribution using 'loads_vsu'")
18 # Pass 3: Model the memory behavior
19 # Pass 3.1 Define the memory model
20 model = "L1 = 33%", "L2 = 33%", "L3 = 34%"
21 synth.add_pass("Generate addresses
22                 according to 'model'")
23 # Pass 4: Init registers
24 synth.add_pass("Init registers to 0b01010101")
25 # Pass 5: Init immediate operands
26 synth.add_pass("Init immediates to 0b01010101")
27 # Pass 6: Model instruction level parallelism
28 synth.add_pass("Set instruction dependency
29                 distance randomly")
30 # Generate the 10 micro-benchmarks and save them
31 for idx in range from 1 to 10:
32   ubench = synth.synthesize() # Apply the passes
33   ubench.save("./example-%s.c"%(idx)) # Save
```

Figure 2: MicroProbe pseudo-code script that generates 10 micro-benchmarks consisting of an end-less loop with 4K load vector instructions that hit equally the three levels of the cache hierarchy. The highlighted parts in gray show how the micro-architecture information is queried to assist the micro-benchmark generation process.

improved features that we identified above. We focus the discussion on the features that are used in the case studies presented in Sections 4, 5 and 6. Other features implemented in MicroProbe are not included due to space limitations.

We guide the discussion using the MicroProbe script example shown in Figure 2. In this example, the user defines a policy to generate micro-benchmarks for the POWER7 micro-architecture (lines 2–3). The micro-benchmarks generated will be composed by an end-less loop of 4K instructions (lines 9–10). The instructions will be load vector instructions (lines 11–17) that hit equally to the three levels of the cache hierarchy (lines 18–22). The registers and immediate operands of the instructions will be initialized to a constant value (lines 23–26) and the dependency distance between the instructions will be assigned randomly (lines 27–29). Finally, the benchmarks synthesizer is invoked 10 times to generate 10 different micro-benchmarks (lines 31–33).

2.1. The Architecture module

The three main functionalities implemented in the *Architecture module* are the following: the PowerPC *ISA definition*, the POWER7 *Micro-architecture definition* and the set-associative cache model (a *Micro-architecture analytical model*).

The first functionality, the description of the PowerPC ISA, is used in the example to filter the load instructions of the ISA (lines 12–13 in Figure 2). The second, the POWER7 micro-architecture definition that provides the mapping between instructions and micro-architecture components stressed, is used in lines 14–16 of the example to select only the loads that stress the Vector Scalar Unit (VSU). The last functionality, the analytical set-associative cache model, is used to statically ensure a specific distribution among the memory hierarchy levels (lines 18–22). The following sections present the details of these three main modules of the *Architecture module*.

2.1.1. ISA definition module: This module implements the capability to generate assembly code for the target ISA. It leverages the format and the valid operands for each instruction of the ISA plus a rich set of semantic information for each of them. This includes the instruction type (e.g. load, store, vector, int, float or branch), the length of the operands of the instruction, if the instruction is executed conditionally, the privilege level required for the instruction, if the instruction is a data pre-fetch instruction, the registers used/defined by the instruction, the binary codification of the instruction, etc. This information is extensible and accessible by the user to perform any action based on it. For instance, one can select only the load instructions as shown in line 13 of Figure 2.

The ISA definitions are supplied to MicroProbe using readable text files. These definition text files are constructed using the information from ISA definition manuals. For this work, we implemented the definition text files for the Power ISA v2.06B [36]. This text-file based ISA definition approach provides an extra level of flexibility and adaptability. For instance, the user can add/remove instructions from the ISA and re-execute the very same MicroProbe script without requiring the modification of the MicroProbe internals.

2.1.2. Micro-architecture definition module: This module provides the information related to the specific micro-architecture implementation. From the architecture implementation point of view, this refers to the micro-architecture components and their hierarchy (functional units/sub-units), the cache hierarchy characteristics, the layout of the micro-architecture units (area, floor-plan information, etc.), the performance counters related to each micro-architecture component, etc. From the ISA point of view, this information includes the latency, throughput, power or EPI (energy-per-instruction) of the instructions. Moreover, the mapping between the instructions and the micro-architecture components they stress is also provided. For instance, the lines 14–16 of Figure 2 show how this information is used to select the instructions stressing the VSU unit. This rich set of low-level information, which simplifies the micro-benchmark generation task, is one of the new features that differentiates MicroProbe from all previous work.

Automatic bootstrap support. Similar to the ISA definition, the micro-architecture definition is supplied to MicroProbe using text files. This increases the portability of the framework. However, the process of setting up a complete micro-architecture definition is a time-consuming task that can still limit the portability of the framework. The reason is that all the details in the micro-architecture definition must be re-defined for each micro-architecture implementation. MicroProbe avoids this problem by implementing a bootstrap process that automatically completes a partial micro-architecture definition.

The following information is required to start the bootstrap process: (a) the micro-architecture functional units within the system. This includes their basic information (e.g. name) and their associated performance counters; (b) the definition of the 'IPC' property of the system (the performance counter-based formula); and (c) the ISA implemented in the micro-architecture.

The bootstrap process then generates two micro-benchmarks for each instruction of the ISA. The first micro-benchmark is an end-less loop consisting of 4K instances of the instruction with a chain of dependencies across any two consecutive instructions. The second micro-benchmark is similar to the first one except that there are no dependencies. Both micro-benchmarks are executed and the performance counters related to the functional units and IPC are read. From these readings, MicroProbe derives the instruction latency and the units that are stressed. MicroProbe proceeds similarly with the second micro-benchmark to derive the throughput and confirm the functional units stressed.

In order to bootstrap the EPI or the average sustained power metrics, MicroProbe also reads the power sensors. MicroProbe uses the micro-benchmark version without dependencies to bootstrap these metrics. The micro-benchmarks generated use random values to initialize registers, immediate values and memory regions. This minimizes the possible data switching effects, allowing fair comparison between instructions [44]. The case study presented in Section 5 provides more insights about the automatically bootstrapped per-instruction EPI information.

2.1.3. Micro-architecture analytical models: Dynamic micro-benchmark properties are usually ensured by performing time-consuming design space explorations (DSEs). This process looks for the correct micro-benchmark generator input parameters to generate a micro-benchmark that satisfies the target dynamic properties. This process needs to evaluate each possible solution generated; therefore, it can be a practical limitation in real execution environments. However, it is known that under constrained conditions and detailed micro-architecture information, one can define analytical models to statically ensure dynamic properties of micro-benchmarks [14]. Therefore, the use of analytical models speeds up the micro-benchmark generation process, avoiding the time-consuming DSEs.

The level of detail provided in the *Micro-architecture definition* module enables the implementation of micro-architecture analytical models within MicroProbe. For instance, MicroProbe implements an analytical memory model for traditional set-associative cache hierarchies. We use it to generate in one step the micro-benchmarks with specific memory activities used in Section 4. The following section provides an overview of the rationale of this novel analytical memory model.

Set-Associative cache model: Previous work on micro-benchmark generation models the memory behavior by generating particular stride patterns that walk through pre-allocated memory [33]. It assumes that different stride values lead to different hit/miss ratios. Then, if a particular hit/miss ratio is required, a design space exploration (DSE) can be done to find the number of patterns, including their distribution and their strides. This would generate a targeted memory activity. Our modeling method avoids the need to perform a DSE and statically ensures the requested activity in each level of the cache hierarchy. The method is based on the following two observations:

First, with appropriate information —provided by the *Micro-architecture definition* (See Section 2.1.2)— it is possible to know and control the set used on each cache level when a memory operation is executed. For instance, Figure 3a shows how main memory blocks are mapped into a 4-way set associative cache. If we generate addresses within blocks 0, 128 or 256, we know that the data will be placed in Set 0.

Second, it is possible to ensure a hit or a miss in a particular cache level if enough accesses are generated. Taking into account the same example shown in Figure 3a, if we generate more than 4 consecutive

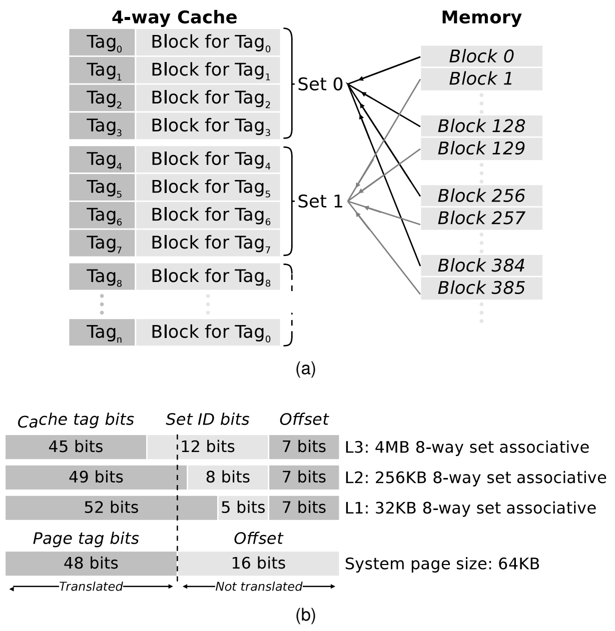


Figure 3: (a) Set-associative cache diagram. (b) Address fields at each level of the cache hierarchy and at the operating system level on a POWER7 platform.

memory requests hitting the set 0 within an end-less loop, we can ensure that the loop will enter a steady state where all accesses will miss. The memory requests should be randomized to minimize the interferences of the hardware pre-fetchers. On the contrary, if we generate 4 or less accesses hitting the set 0, the loop will always hit.

From these observations we can derive that it is possible to generate a sequence of memory accesses to ensure a particular distribution of the requests among the different levels of the cache hierarchy. For that purpose, we assign disjoint sets —sets that do not conflict— to each memory hierarchy level. We then generate the adequate number of accesses for each cache level. This is possible because: (a) MicroProbe provides to the user full control on the code being generated, and (b) the micro-architecture definition contains the required information to infer the set fields of each cache level. Figure 3b shows the set fields of each cache hierarchy level on our experimental platform.

This memory modeling method is used to apply the power modeling methodology presented in Section 4. This power modeling methodology requires several micro-benchmarks covering a wide range of memory activities. In this situation, being able to statically ensure memory activity rates reduces the time required to generate the micro-benchmarks.

2.2. The code generation module

The code generation module contains the micro-benchmark synthesizer. The micro-benchmark synthesizer is the core component of any micro-benchmark generation framework because it is in charge of driving the code generation process. Previous work [4, 33] identified that the code generation process requires a minimum number of steps to define the final behavior of the micro-benchmarks generated. These steps are the following: (1) define the program skeleton (e.g. the size of basic blocks; number of threads, etc.); (2) define the instruction distribution; (3) model the memory behavior (i.e. define how the memory is accessed); (4) model the branch behavior (i.e. control the level of speculation); (5) model the instruction level parallelism (ILP) via register allocation (i.e. define the dependency distance between instructions). This step wise approach has been observed to be the common method to define the properties of the micro-benchmarks generated.

We designed the micro-benchmark synthesizer of MicroProbe to work in a compiler-like fashion. The rationale is that this design provides the flexibility and extensibility required to adapt the micro-benchmark generation process to the user's requirements. This differs from prior work, where the transformations and the sequence of steps are fixed and tailored to solve specific problems. The example script of Figure 2 shows how the user defines the sequence of transformations (i.e. their type and their order) required to generate the micro-benchmarks. We call these transformation steps *passes*.

Within MicroProbe, new passes can be added and sorted at user's will, making the framework extensible and adaptable. Many basic passes, like the ones in the example in Figure 2, are already available in our framework. This forms a general repository of passes for designing complex micro-benchmark generation policies. To name a few, we have implemented a pass to set up an end-less loop with *n* instructions (line 8 of Figure 2), a pass to generate a given instruction distribution (line 17 of Figure 2) and a memory pass that ensures a given memory activity (See Section 2.1.3). Several other passes to model branch behavior, initialize values, etc. are also implemented. We refer the reader to previous work on micro-benchmark synthesizers [18, 20, 21, 33, 40] to read about other possible transformation passes that can be implemented on top of MicroProbe.

We show the importance of having a compiler-like design explaining a possible real world example. Let's suppose that we have a computational kernel and we want to test the effect of certain transformations on it. We set up a MicroProbe script to generate the baseline code —i.e. the initial sequence of instructions comprising the kernel. We may then want to evaluate the effect on performance of unrolling the loop or the effect on power of using a *load immediate* and an *add* instruction instead of two *add immediate* instructions. For that purpose, we simply copy the original MicroProbe script that generates the computational kernel and then add the extra passes to apply the transformations. This level of adaptability is enabled by the pass-based design of the micro-benchmark synthesizer.

2.3. Design space exploration module

Design space explorations (DSE) have become mandatory to understand the performance of computer architectures due to their increase in complexity. In addition, DSE are required to find micro-benchmarks that fulfill a set of dynamic properties that cannot be ensured statically. DSE support is therefore a basic functionality that any productive micro-benchmark generation framework should have.

MicroProbe provides generic DSE support to be able to implement different customizable search strategies within the design space. For instance, MicroProbe currently supports exhaustive searches, genetic algorithm (GA) searches and user-defined searches. This is in contrast to previous work, which only provided GA search support [20, 21, 33, 40]. Thus, MicroProbe provides an adaptive framework for performing DSEs.

Moreover, the fact that DSE support is integrated within the same framework is also beneficial. Previous work decoupled the micro-benchmark synthesizer component from the search driver component, thus losing possible synergies between these components. MicroProbe integrates both functionalities into the same framework. This allows, for instance, the definition of user-guided drivers that query the micro-architecture information in order to guide the search. In Section 6 we use the integrated DSE support of MicroProbe to generate a max-power stressmark. The search driver we define uses the per-instruction EPI information and the mapping between instructions and the functional units to focus the search on certain parts of the design space.

Feature	**Micro-Probe**	**Previous Work**
ISA queries		
- *instruction type*	✓	✓
- *operand length*	✓	Manual[1]
Micro-architecture queries	✓	
- *functional unit*	✓	Manual[1]
- *latency*	✓	Manual[1]
- *throughput*	✓	Manual[1]
- *energy per instruction (EPI)*	✓	Manual[1]
- *average instruction power*	✓	Manual[1]
Micro-architecture models		
- *Set associative cache model*	✓	No
Code generation		
- *Skeleton definition pass*	✓	✓
- *Instruction definition pass*	✓	✓
- *Basic memory modeling pass*	✓	✓
- *Branch modeling pass*	✓	✓
- *ILP definition pass*	✓	✓
- *Configurable passes*	✓	No
Design space exploration		
- *Integrated*	✓	No
- *GA-based search*	✓	✓(External tool)
- *Exhaustive search*	✓	✓(Manually)

[1]The user manually or using an external tool has to obtain the information to pass the appropriate inputs to the code generator.

Table 1: Summary of the novel MicroProbe features and their implementation in previous work.

2.4. Summary of MicroProbe features

Table 1 summarizes the micro-benchmark generation features included in MicroProbe and their implementation in prior work. MicroProbe provides detailed architecture related information such as queries about ISA and micro-architecture information. This depth of architecture-related information is not offered in previous micro-benchmark generation frameworks. Some prior work includes limited instruction type semantics. However, simple queries like functional unit information (lines 15–16 in Figure 2) or instruction latency information require the user to obtain the information manually. In the end, the lack of this integrated low-level micro-architecture semantics diminishes the benefits of having an automatic micro-benchmark generator.

MicroProbe implements micro-architecture models such as the set associative cache model. As far as we know, this feature is not found in previous work. MicroProbe does provide the basic support for code generation, as in prior work. In other words, it supports at least the minimum set of transformation passes that define the behavior of the micro-benchmark generated. MicroProbe goes one step further by improving the flexibility of the code generation support by allowing the passes to be configured. Finally, regarding the DSE support, MicroProbe integrates such support within the same framework whereas previous work uses external tools or manual setups to perform DSEs.

3. Experimental Framework

The experimental platform is an IBM BladeCenter PS701 system. The system has one POWER7 processor running at 3.0 GHz and 32 GB of DDR3 SDRAM running at 800 MHz. The IBM POWER7 processor is an eight-core chip where each core can run up to four threads. Each core has 32KB first level, 256KB second level and 4MB third level data cache. A detailed specification of the architecture is available elsewhere [42]. The platform runs RHEL 5.7 OS with linux kernel version 3.0.1. This version provides the standard PCL API [17] to access hardware performance counters.

The platform implements the EnergyScale architecture [19] that allows the users to gather the power consumption of the processor via the Flexible Support Processor (FSP). The FSP accesses the micro-controller called Thermal and Power Management Device (TPMD) to perform the sensor readings. Both devices, the FSP and the TPMD, are managed by the BladeCenter chassis Management Module (MM).

We use an in-house software to monitor all the sensors required for the experiments. The software can sample sensors at 1-ms granularity. Power measurements are in the granularity of milliwatts, whereas the temperature measurements are in degrees celsius. We also gather performance monitoring counter (PMC) traces to account for different activity of the micro-benchmarks and the SPEC CPU2006 benchmarks that are executed. Power and performance counter traces are then analyzed and plotted using the POTRA framework [6].

Micro-benchmarks are deployed as one copy per hardware thread context that is available on the configuration. For example, in a 2-way SMT 6-core configuration, we deploy 12 copies of the micro-benchmark. We pin each copy to a logical CPU to avoid thread migrations. We run the micro-benchmarks for 10 seconds which helps us to shorten the data gathering process while still providing valid —and stable— power and performance counter values. Similarly, we also execute the SPEC CPU2006 [25] benchmark suite for model validation purposes. The SPEC CPU2006 benchmarks are run to completion. All power results presented in this work are in normalized form to avoid disclosure of absolute values.

4. Bottom-up CMP/SMT aware counter-based processor power model

One important area where MicroProbe provides special value is in the task of generating empirical counter-based power models. Such power models are of key interest because they provide a quick path to estimate run-time power consumption without the need to rely on direct measurement devices [5, 27]. Counter-based power models have not only been used to model the power consumption of the processor [5, 29, 35, 43], they also have been useful to predict the consumption of the rest of the components in the system [10, 11].

In particular, *bottom-up* counter-based power modeling methodologies have been shown to be a competitive approach [7]. Besides accuracy and generality [8, 9], these types of models provide a fine-grained granularity, sometimes allowing per functional unit breakdowns [27]. Although we do not focus this work to reach such low level of decomposability, we present a method to generate bottom-up counter-based power models for SMT/CMP processors such as the POWER7.

Bottom-up processor power models predict the overall power consumption of the processor as the sum of the power consumption of different power components. These power components are usually associated with micro-architecture components [8, 9, 27]. This allows the users to derive the power breakdown across these components. This adds insight on power behavior across workloads and individual components within the processor. In a CMP/SMT system, this capability is useful in discerning the power consumption of each core or hardware thread. In addition, the power-related effects of enabling the SMT logic or enabling/disabling cores can be easily quantified.

Previous methods of bottom-up processor power modeling were applied to processors that lack the level of parallelism and complexity of the POWER7. In [27], a bottom-up power model of a Pentium 4 is presented. In [7–9], the authors model a dual-core processor without SMT. In contrast, we model a highly parallel processor such as the POWER7, with 8 cores and up to 4-way SMT capabilities.

Bottom-up counter-based modeling methods require micro-benchmarks that stress different micro-architecture functional units at different levels. This is needed to estimate individual contributions to the overall power consumption [8, 9, 27]. In this context, a common rule of thumb is to use a very broad range of power contexts for training the model. This is known to result in a more general and accurate model. This implies a rather time-consuming task of generating a huge set of micro-architecture aware micro-benchmarks. This requirement delayed the application of bottom-up modeling methods on current architectures. The main reason was the lack of micro-benchmark generation frameworks like MicroProbe that have micro-architecture semantics.

We use MicroProbe to generate the rich set of micro-benchmarks shown in Table 2. We generate micro-benchmarks that stress different combinations of functional units at different levels (IPCs) by using the micro-architecture information and the DSE GA-based support implemented in MicroProbe. The functional units of the POWER7 processor that we stress are: the fixed point unit (FXU), the load store unit (LSU) and the vector scalar unit (VSU). We also generate micro-benchmarks stressing the memory hierarchy at different levels. We stress the four levels of the memory hierarchy: the first-level cache (L1), the second-level cache (L2), the third-level cache (L3) and the main memory (MEM). In this process, the analytical micro-architecture memory model of MicroProbe (See Section 2.1.3) removes the necessity to perform a DSE for each memory activity we target. Finally, micro-benchmarks with random activities are also generated in order to enrich the training set.

Notice that hand-crafting —and verifying— this micro-benchmark suite is normally a very time-consuming effort. With MicroProbe we are able to do it in a few hours without any human intervention. The next section explains the modeling methodology that uses these micro-benchmarks to produce a SMT/CMP aware bottom-up counter-baser processor power model.

4.1. SMT/CMP aware bottom-up modeling methodology

We apply the bottom-up methodology shown in Figure 4 to model the processor. This methodology ensures the decomposability of the model because it models the power consumption of the different processor power components defined separately. We define the following four power components: (a) the dynamic power consumption, i.e. the power related to the activity of the hardware contexts running on the system; (b) the SMT effect, i.e. the power contribution of enabling the SMT logic of the cores; (c) the CMP effect, i.e. the power contribution of enabling multiple cores on the system; and (d) the uncore power contribution, i.e. the constant power contribution of having activity on the processor. Moreover, there is the workload independent power consumption, which is the power consumption of the processor when there is no activity.

Name	Units stressed[1]	#	Description	MicroProbe features
Simple Integer	FXU or LSU	35	Mix of simple integer instructions (can be executed by the LSU or FXU units) with IPCs from 0.5 to 4 in steps of 0.1.	ISA & uarch queries & DSE GA support
Complex Integer	FXU	11	Mix of complex integer instructions (only can be executed by the FXU unit) with IPCs from 0.1 to 1.1 steps of 0.1.	"
Integer	FXU, LSU	12	Mix of integer instructions with IPCs from 0.10 to 1.20 in steps of 0.1.	"
Float/Vector	VSU	14	Mix of vector, float and decimal instructions with IPCs from 0.1 to 1.4 in steps of 0.1.	"
Unit Mix	VSU, FXU, LSU	20	Mix of all kind of instructions (non memory, no branch) with IPCs 0.1 to 2 in steps of 0.1.	" "
L1 ld	LSU, L1	10	Random mix of load instructions hitting the L1.	ISA queries & uarch model
L1 ld/st	LSU, L1, L2	10	Random mix of load/store instructions hitting the L1.	"
L1L2a	LSU, L1, L2	10	Random mix of load/store instructions 75% hitting the L1 and 25% hitting the L2.	"
L1L2b	LSU, L1, L2	10	Random mix of load/store instructions 50% hitting the L1 and 50% hitting the L2.	"
L1L2c	LSU, L1, L2	10	Random mix of load/store instructions 25% hitting the L1 and 75% hitting the L2.	"
L1L3a	LSU, L1, L2, L3	10	Random mix of load/store instructions 75% hitting the L1 and 25% hitting the L3.	"
L1L3b	LSU, L1, L2, L3	10	Random mix of load/store instructions 50% hitting the L1 and 50% hitting the L3.	"
L1L3c	LSU, L1, L2, L3	10	Random mix of load/store instructions 25% hitting the L1 and 75% hitting the L3.	"
L2	LSU, L1, L2	10	Random mix of load/store instructions hitting the L2.	"
L2L3a	LSU, L1, L2, L3	10	Random mix of load/store instructions 75% hitting the L2 and 25% hitting the L3.	"
L2L3b	LSU, L1, L2, L3	10	Random mix of load/store instructions 50% hitting the L2 and 50% hitting the L3.	"
L2L3c	LSU, L1, L2, L3	10	Random mix of load/store instructions 25% hitting the L2 and 75% hitting the L3.	"
L3	LSU, L1, L2, L3	10	Random mix of load/store instructions hitting the L3.	"
Caches	LSU, L1, L2, L3	10	Random mix of load/store instructions 33% hitting the L1, 33% hitting the L2 and 34% hitting the L3.	"
Memory	LSU, L1, L2, L3, MEM	20	Random mix of load/store instructions missing in all levels of the cache hierarchy.	"
Random	Unknown	331	Random micro-benchmarks.	ISA queries

[1]FXU: fixed point unit (integer), LSU: load store unit (memory operations) and VSU: vector scalar unit (vector, float and decimal operations).
L1: L1 cache, L2: L2 cache, L3: L3 cache, MEM: Main memory

Table 2: Micro-benchmarks automatically generated using MicroProbe. They cover a broader scope of possible processor activities in order to increase the accuracy of the models generated. They share a common skeleton: a 4K endless loop with the required instructions to stress particular functional units.

The bottom-up modeling methodology used, introduces two new components when compared to previous bottom-up modeling methods [8, 27]. The new components are the SMT effect and the CMP effect components.

The SMT effect component models the extra power required when SMT is enabled. We observed empirically that two workloads exhibiting the very same overall core activity consume a different amount of power depending on whether SMT is enabled or disabled. The reason is that the extra control logic that in operation when SMT is enabled consumes additional power. This effect is independent of whether 2-way SMT or 4-way SMT is enabled.

The second new component, the CMP effect, models the change of uncore power consumption depending on the number of cores enabled. This power consumption changes due to the different usage of the shared components when different number of cores are used. For instance, the 32MB last level cache of the POWER7 is partitioned into eight equally sized slices, one for each core. When a core is not used, the last level cache slice of that core is used only as a victim cache of the other slices, changing its usual power behavior. The CMP effect captures the specific conditions in power consumption that depend on the number of cores enabled.

The addition of these two variables is crucial to increasing the accuracy of the models. They are not directly related to the actual activity in the processor like performance counters. However, they affect how the activity is being performed as well as the power status of different micro-architecture components. Models without these two input variables —the SMT enabled and the number of cores enabled (*#cores*)— exhibit large errors in the predictions and show inconsistencies across the different SMT and CMP modes of operation. The rest of the section explains the details of each of the four modeling steps shown in in Figure 4.

Step 1: Model a Single Hardware Context: We model a single core in single-threaded (SMT-1) configuration using the bottom-up modeling method detailed in [8]. In brief, we define the FXU, VSU, LSU, L1, L2, L3 and MEM as the power components of the processor cores. We assign a performance monitoring counter (PMC) based formula for each of these components. A sequence of linear regressions is then performed to model separately the power contribution of each of these power components. This is possible because the specifically designed training set covers a wide set of scenarios that stress different units at different utilization rates [8]. The model intercept is then calibrated using the random micro-benchmarks to avoid underestimating the power when only particular units are stressed [8, 49]. The result of this process is a bottom-up power model for a single core in SMT-1 configuration. The dynamic component of the model, the part that it is dependent on the PMCs (Dynamic Power in Figure 4), is defined as:

$$\begin{aligned} P_{dyn} &= FXU_{pmcs} \times W_{fxu} + VSU_{pmcs} \times W_{vsu} \\ &+ LSU_{pmcs} \times W_{lsu} + L1_{pmcs} \times W_{l1} + L2_{pmcs} \times W_{l2} \\ &+ L3_{pmcs} \times W_{l3} + MEM_{pmcs} \times W_{mem} \end{aligned}$$

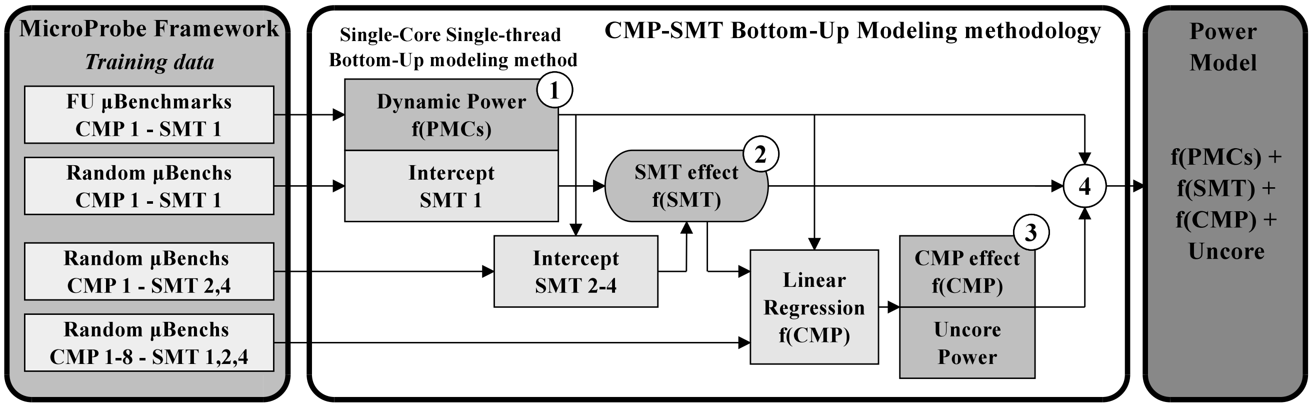

MicroProbe Framework
Training data
FU µBenchmarks CMP 1 - SMT 1
Random µBenchs CMP 1 - SMT 1
Random µBenchs CMP 1 - SMT 2,4
Random µBenchs CMP 1-8 - SMT 1,2,4
Single-Core Single-thread Bottom-Up modeling method
CMP-SMT Bottom-Up Modeling methodology
1
Dynamic Power f(PMCs)
Intercept SMT 1
2
SMT effect f(SMT)
Intercept SMT 2-4
Linear Regression f(CMP)
3
CMP effect f(CMP)
Uncore Power
4
Power Model
f(PMCs) + f(SMT) + f(CMP) + Uncore

Figure 4: Proposed SMT/CMP aware bottom-up modeling methodology. (1) A single core —a single hardware context— is modeled; (2) the effect of enabling SMT is estimated; (3) the effect of enabling cores —the CMP effect— and the uncore power consumption are estimated; (4) the final model is defined as the sum of the power consumption of each hardware context, the SMT effect of each core with SMT enabled, the CMP effect and the uncore power consumption.

The non-dynamic component (intercept SMT-1 in Figure 4) is used in the next step to compute the SMT effect.

Step 2: Model the SMT Effect: As stated previously, we observed that the power consumption is higher when SMT is on. We simplify the modeling of this behavior by assuming that the power consumption increases by a fixed value when SMT is activated. We therefore model the SMT effect as a constant value, which is defined as:

$$SMT_{effect} = Intercept_{SMT2-4} - Intercept_{SMT1}$$

where the SMT effect value is the difference of the uncore power consumption between a model trained using SMT enabled data (intercept SMT-2–4) and the model trained using SMT disabled data (intercept SMT-1).

Step 3: Model the CMP Effect and the uncore power: To model the CMP effect and the uncore power, we apply the dynamic and the SMT effect models defined in steps 1 and 2 to the random micro-benchmarks executed in all SMT and CMP configurations (See step 3 of Figure 4). After applying the model, we obtain the residuals of the predictions. These residuals, which exhibit a positive correlation with the number of cores enabled, can be interpreted as the power consumption related to the change in the number of cores plus the uncore power. We therefore model the residuals as a function of the number of cores enabled (*#cores*) using a linear regression of the form $a \times x + b$. The intercept of the obtained regression (i.e. b) is assumed to be the uncore power consumption (P_{Uncore}) whereas the $a \times x$ component is assumed to be the CMP effect ($CMP_{effect} \times \#cores$).

Step 4: Combine the models: We combine all the modeled power components to obtain the final bottom-up power model. The model is therefore defined as:

$$\begin{aligned} P_{cpu} &= \sum_{k=1}^{\#threads} P_{dyn_k} + \sum_{k=1}^{\#cores} SMT_{effect} \times SMT\_enabled_k \\ &+ CMP_{effect} \times \#cores + P_{Uncore} \end{aligned}$$

which is the addition of the power consumption of each hardware thread enabled on the platform (step 1), the SMT effect of the cores with SMT enabled (step 2), the CMP effect as a function of the number of cores enabled and the uncore power consumption (step 3).

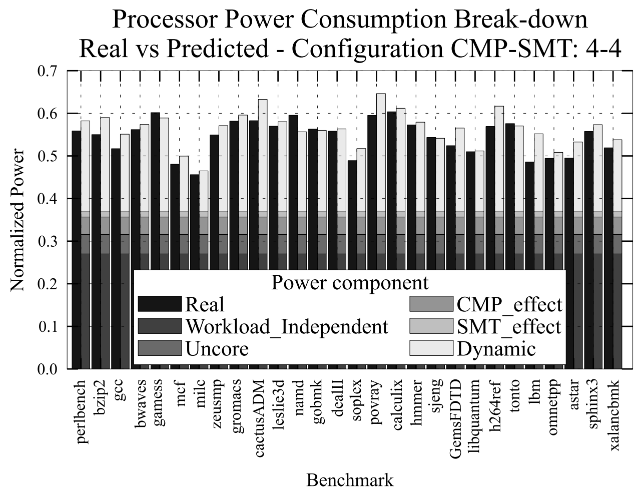
4.1.1. Model Validation: Figure 5a shows how the model is able to track the power consumption of the SPEC CPU2006 on a 4 core, 4-way SMT configuration. The dynamic power consumption varies with the workload whereas the rest of components remain constant because they depend on the processor SMT/CMP configuration. This power consumption breakdown is only possible because of the bottom-up modeling methodology. Top-down modeling methods [5, 11, 23, 41] model the processor as a black box. They are able to perform per-core power estimations by gathering per core performance counters. However, they do not provide the same insights [7].

Figure 5b shows the percentage average absolute prediction error (PAAE) [10] of the proposed bottom-up (BU) model when compared to actual measured power of the SPEC CPU2006 workloads for all the configurations studied. The maximum PAAE is around 4% and most of the values are below 2.3%, which is the average PAAE. These results validate that the novel SMT/CMP aware bottom-up modeling method is able to model different SMT/CMP configurations accurately.

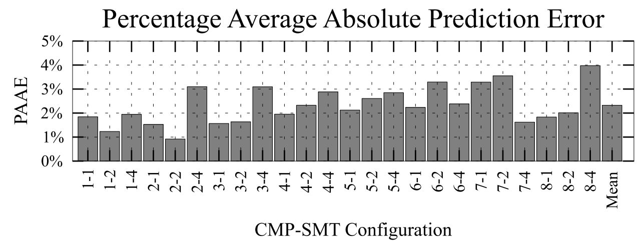
There is, however, a small trend that shows higher errors for higher number of cores. This might be related to the CMP and SMT attributes, which we modeled assuming a linear relation. This linear approximation is necessary to help us to create the bottom-up hierarchical CMP/SMT power model. The implicit assumption of linear dependence of these attributes on power is an approximation of what is most likely a non-linear model. For example, if the real values follow a monotonic convex/concave curve, a linear approximation will yield an error function that causes absolute error to first increase and then decrease, as seen in Figure 5b.

4.1.2. Comparison to other Models: We compare our bottom-up (BU) model against a set of top-down (TD) models [7] in order to bring out the benefits of the bottom-up modeling approach. TD modeling methodologies use parameter selection techniques to select the model inputs and then they apply a single multiple linear regression to model the entire processor. These models do not require specifically designed micro-benchmarks. They are therefore a popular solution due to their simple generation. However, they do not provide the same accuracy and generality as the bottom-up models.

We generate three TD models using the same inputs of our bottom-up (BU) model for fairness: namely, the functional unit performance counters, the numbers of cores enabled and the SMT mode. The models are named after the training set used to generate them: the



(a)



(b)

Figure 5: (a) Processor power consumption breakdown of the SPEC CPU2006 on a 4 core, 4-way SMT configuration. (b) Percentage average absolute prediction error (PAAE) of the model for all the configurations analyzed.

micro-architecture aware micro-benchmarks (TD_Micro), the random micro-benchmarks (TD_Random), and the SPEC benchmarks (TD_SPEC). TD_SPEC is therefore the optimistic model because it has been trained using the validation set. As explained previously, the BU model is trained using all the micro-benchmarks, namely the micro-architecture aware micro-benchmarks and the random micro-benchmarks.

Figure 6 shows the average PAAEs on the SPEC CPU2006 with respect to the models generated for each configuration studied. All the models show similar trends confirming that each training set covers enough power contexts to model the SPEC CPU2006 suite accurately. In general, the models are consistent across the different SMT/CMP configurations. Only the TD_Micro shows a consistently higher error for 2-way SMT configurations. In any case, all the models show acceptable results on average.

The last columns of Figure 6 show mean PAAEs around the 2–4% range. When compared to the optimistic model (TD_SPEC), the rest of the models show less than 2 percentage points of difference. These accurate predictions are enabled by the inclusion of the SMT and CMP variables to the models. The proposed BU model outperforms the rest, being the one closer to the optimistic TD_SPEC model.

4.1.3. Model Validation on Extreme Cases: Although there is not a clear difference in accuracy between the different modeling methods for general workloads, there is a significant difference when extreme cases are considered. We consider different extreme cases such as high and low integer (FXU) or vector activity (VSU), only L1 loads or only memory activity. Although we call these cases *extreme*, these types of activities are actually quite common in applications over short periods of time. For instance, consider the case of a highly optimized vector loop accessing only the first level cache. In such a case, the processor will show a period with only high IPC vector activity. Similarly, when the processor copies data from main memory to a local array, only main memory activity will be exhibited.

Figure 7 shows the PAAEs of the models for the extreme activity cases considered. The models trained using micro-architecture aware micro-benchmarks (i.e. the TD_Micro and the BU models) are capable of modeling these situations accurately, whereas the models trained using general workloads exhibit high errors. For instance, the TD_Random model shows a 62% PAAE for the FXU High case. This is because the models trained using general workloads are biased towards the *normal* activities they exhibit. In contrast, the models trained using micro-architecture aware training sets show similar accuracy levels across general and extreme workloads.

This observation highlights the benefits of generating micro-architecture-centric models like the bottom-up model instead of the workload-centric models like the top-down models. A framework like MicroProbe, capable of generating micro-architecture aware micro-benchmarks, is therefore essential for facilitating the generation of micro-architecture aware training sets.

4.1.4. SMT/CMP Effects on Power Consumption: In this section, we use the decomposability capability provided by the bottom-up model to analyze how the SMT/CMP configuration affects the distribution of power consumption. Notice that this level of insight is not possible using top-down models [7]. Figure 8 shows the average percentage power consumption breakdown for the SPEC CPU2006 for each configuration analyzed.

From the SMT point of view, changing the SMT configuration increases the percentage of dynamic power consumption of the processor by about 10 points. At the same time, it decreases the workload independent power component by an identical amount. The reason is twofold: (a) the more hardware contexts are enabled, the more dynamic power is consumed due to the increase of ILP within the cores; (b) this increase in dynamic activity exceeds the overhead of enabling the SMT feature (SMT_effect in Figure 8), which we found to be minimal (<3% in all the cases).

The components that do not depend on the CMP parameter —the workload independent and the uncore components— account for up to 85% of the overall power consumption in the lowest configuration (i.e. 1 core, 1-way SMT configuration). This percentage is reduced to 50% as we increase the number of hardware contexts (8 cores, 4-way SMT configuration). This is mainly due to the increase of the dynamic component. We also observe that the power breakdown remains comparable when a minimum of 4-cores are enabled. Beyond that point, adding extra cores results in a similar increase of dynamic and non-dynamic power consumption, suggesting that the shared resources are already fully utilized. For instance, in Figure 8, going from 1–1 to 2–1 CMP–SMT configuration reduces the workload independent and uncore power consumption from 85% to 77%. However, going from 7–1 to 8–1 only reduces these components by 1 percentage point, from 62% to 61%.

In summary, we present a novel bottom-up power modeling methodology capable of modeling the SMT/CMP features of current architectures. We show how the model generated using this methodology outperforms existing approaches for normal and extreme workloads. The basis of the model is a complete micro-architecture aware

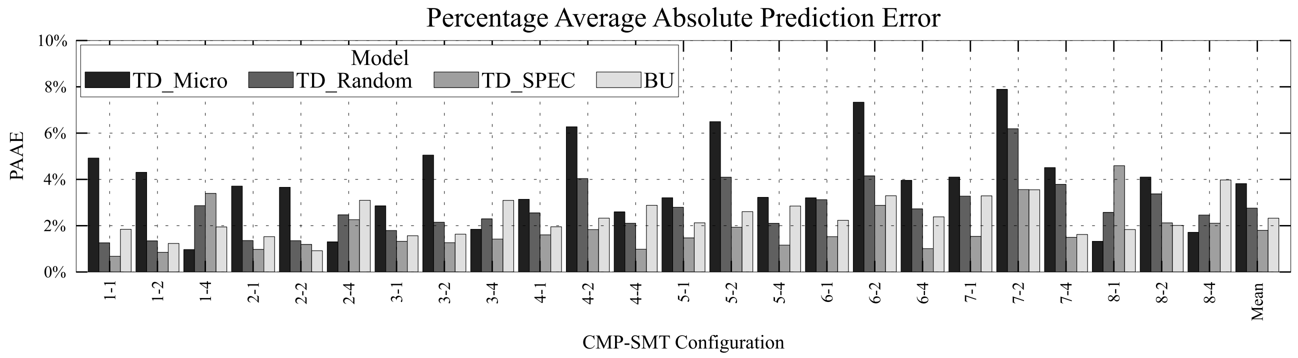


Figure 6: Percentage average absolute prediction errors of the models generated when compared to actual measured power of the SPEC CPU2006 workloads for all the configurations analyzed.

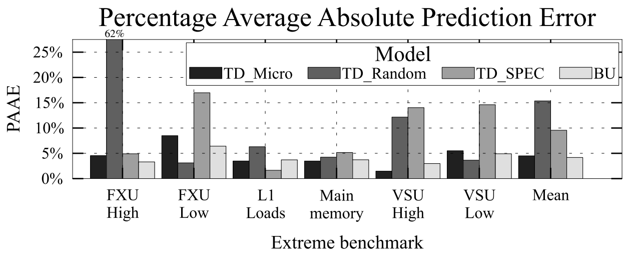


Figure 7: Percentage average absolute errors of the models generated for all configurations on the extreme situations analyzed.

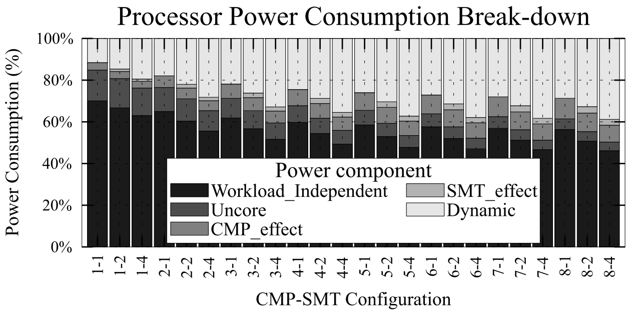


Figure 8: Average per component power consumption breakdown of the SPEC CPU2006 benchmarks for all the configurations of the processor.

training set systematically generated using MicroProbe. Finally, we use the extra information provided by the model to study the SMT/CMP effects on power consumption.

5. POWER7 energy-based instruction taxonomy

Another important area where MicroProbe is useful is in the low-level characterization of architectures. MicroProbe's bootstrap process explained in Section 2.1.2 automatically gathers per instruction micro-architecture information such as latency, throughput or energy per instruction (EPI). This information can be analyzed to generate, for instance, an instruction level EPI characterization.

An instruction-level EPI characterization is beneficial in a wide set of situations. For example, this is necessary for understanding tuning opportunities for hardware implementation of instructions or for improving compiler instruction selection algorithms. This characterization is also useful for guiding micro-benchmark generation policies when searching for max-power stressmarks as described in Section 6.

This section develops a taxonomy of the POWER7 instructions based on energy per instruction (EPI) and processor activity characteristics. We use the unit-stressing information that is implemented in MicroProbe to classify the instructions in categories based on the functional units that the instructions stress.

The results presented are for the 1-way SMT 8 core configuration. Notice that the EPI values are derived from the overall dynamic processor power consumption. Therefore, they depend on the processor configuration (i.e. number of cores and SMT mode) used. EPI values also depend on the input data used, which we randomized. We do not observe any significant variations in EPI when we randomly change the input values. This agrees with prior published results [44]. However, zero input data values sometimes result in a significant reduction in EPI, up to 40% in some cases.

Table 3 shows the core IPC and normalized EPIs of three instructions for each category defined. Categories are named after the functional units that they stress[2]. We group these categories to simplify the explanation. Category EPI column is normalized to the minimum EPI within the category, whereas global EPI column is normalized to the minimum EPI among all the categories. This simplifies the comparisons between instructions and categories of instructions. The top instruction in each category is the one with higher IPC*EPI product within the category. The other two instructions are selected examples with the same IPC but notable differences in EPI.

Analyzing by categories, we can see that the memory operations with *side-effects* (i.e. those that stress other units apart from the LSU) are the ones with higher EPI. The reason is twofold: (a) these types of instructions activate more functional units. For instance, the vector store operations use the LSU unit (address generation) and the VSU (data propagation of the stored value); and (b) these instructions exhibit a lower IPC —each instruction takes more time to be executed— and as a result, they are less efficient.

Overall, the simple integer operations are the most efficient. The reason is that this type of operations is the most common and therefore the execution is highly optimized. For instance, the load store unit (LSU) of the POWER7 is able to execute these simple integer operations. This allows the program to obtain a high IPC, thus lowering the EPI metric.

Analyzing each category, there are important EPI differences between instructions within the same category. This is observed even in the case where the instructions exhibit the same IPC. For instance, in the VSU category, the *xvmaddadp* instruction has a 75% higher

[2]FXU: fixed point unit (integer), LSU: load store unit (memory operations) and VSU: vector scalar unit (vector, float and decimal operations).

Category	Instr.	Core IPC	Normalized EPI Global	Normalized EPI Category
Functional units				
FXU	*mulldo*	1.40	2.60	2.60
	subf	2.00	1.69	1.69
	addic	2.00	1.00	1.00
LSU	*lxvw4x*	1.68	2.88	1.35
	lvewx	1.68	2.81	1.31
	lbz	1.68	2.14	1.00
VSU	*xvnmsubmdp*	2.00	2.35	1.78
	xvmaddadp	2.00	2.31	1.75
	xstsqrtdp	2.00	1.32	1.00
Simple integer operations				
FXU or LSU	*add*	3.50	1.73	1.49
	nor	3.50	1.58	1.36
	and	3.50	1.16	1.00
Integer memory operations				
LSU and FXU	*ldux*	1.00	5.12	1.21
	lwax	1.00	5.01	1.18
	lfsu	1.00	4.24	1.00
LSU and 2FXU	*lhaux*	1.00	5.51	1.15
	lwaux	1.00	5.29	1.10
	lhau	1.00	4.80	1.00
Vector/Float/Decimal memory operations				
LSU and VSU	*stxvw4x*	0.48	8.36	1.40
	stxsdx	0.48	7.16	1.20
	stfd	0.48	5.97	1.00
LSU and VSU and FXU	*stfsux*	0.48	10.00	1.19
	stfdux	0.48	9.49	1.13
	stfdu	0.48	8.40	1.00

Table 3: Taxonomy of POWER7 instructions based on energy per instruction (EPI) and functional unit usage. Core IPC, category EPI normalized to minimum EPI within the category and global EPI, normalized to *addic* EPI, the minimum shown in the table. The top instruction within each category is the one with higher IPC*EPI product. The other two instructions have the same core IPC but notably different EPI which demonstrates the high power consumption variability between instruction types, even in the same category.

EPI than the *xstsqrtdp* instruction. Similar observations can be seen in the rest of categories. These observations confirm the differences in energy consumption across various instruction types.

In summary, we use MicroProbe to generate an instruction-level EPI characterization of a POWER7 platform. The characterization helps us to understand better the energy trade-offs of the underlying architecture. In particular, the variability seen in the EPI results —even across instructions that use the same functional unit at the same utilization level— highlights the importance of taking into account such variations when generating power/energy aware code.

6. Max-power stressmark generation

Max-power stressmarks are very important for computer architects to make early-stage design decisions such as the design of the package and the power delivery network. Existing systematic max-power stressmarks generators rely on time-consuming genetic algorithm based design space explorations [20, 21, 33, 40]. These solutions use abstract workload models (e.g. %integer, %loads, %stores, etc.) and expert-defined design spaces to make the search of the solution tractable. They therefore provide a 'black-box' solution where intimate knowledge of the architecture is not required. This benefit comes at the expense of losing some discriminating opportunities.

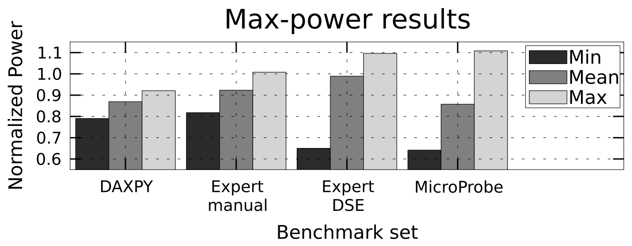


Figure 9: Max, Mean and Min power results for each stressmark set executed. Results are normalized to the maximum power exhibited by one of the SPEC CPU2006 benchmarks during its execution.

For instance, during the selection of instructions, they do not take into account the important differences in power consumption that we have shown in Section 5.

In this section, we show how MicroProbe is used as a 'white-box' framework to help an expert in the process of generating a max-power stressmark in a real measurement context, where the number of design points to explore is a practical limiting factor. In the end, we show that with proper heuristics, the entire process can be fully automated.

We focus this case study on finding the sequence of 6 instructions that when replicated within an endless loop of 4K instructions and executed concurrently on all the available hardware threads maximize the power consumption. The rationale is that basic knowledge in the field suggests that in order to generate a max-power stressmark one should maximize the activity (i.e. maximize the IPC) and maximize the number of functional units used, avoiding pipeline stalls and resource contention (i.e. no dependencies and no memory misses).

Previous work [21] suggests that it would be possible to achieve higher power consumption by executing heterogeneous workloads that stress the different parts of the processor (caches, interconnection network, etc.). We leave the exploration of these options to our future work. We focus this case study on the benefits of using the micro-architecture semantics when generating max-power stressmarks. The fact that we are consistently able to exceed expert level manually generated max-power stressmarks is reassuring.

First, we hand-craft some micro-benchmarks using the *mullw*, *xvmaddadp*, *lxvd2x* instructions. The rationale behind the selection of these instructions is to stress the FXU, the VSU and LSU units using the instructions with a wider data-path (or more complexity) and higher throughput (maximize IPC). This procedure is what a stressmark developer with some expertise in the target micro-architecture would do without support frameworks like MicroProbe. We call this micro-benchmark set as the *Expert Manual* set.

Second, since it is not practical to generate manually all the 540 possible combinations, we use the DSE support of MicroProbe to generate all the combinations of the expert selected instructions automatically. We call this micro-benchmark set as the *Expert DSE* set.

Lastly, instead of relying in our expert to select the instructions, we rely on MicroProbe to select the instruction candidates. We instruct MicroProbe to select the instructions with the highest IPC*EPI product within each functional unit category. This heuristic selects the instructions with a balanced trade-off between EPI and IPC, penalizing instructions with high IPC but low EPI and vice versa. The automatically selected instructions are the top ones shown in the FXU, LSU and VSU categories of Table 3. We call this micro-benchmark set as the *MicroProbe* set.

We execute the three micro-benchmark sets in the three available SMT modes. In addition, various DAXPY kernels with different L1 contained memory foot-prints are also executed. This computational kernel is commonly used as a stressmark. Figure 9 shows the maximum, minimum and average power consumption of each micro-benchmark set. Results are normalized to the maximum power exhibited by one of the SPEC CPU2006 benchmark during its execution.

We observe that with a bit of intuition the expert is able to conceive hand-crafted stressmarks (*Expert manual*) that are as good as the max-power of SPEC CPU2006. However, these stressmarks are still around 10% below the one achieved by the *Expert DSE* set —even though they use the same instruction types and exhibit the same IPC.

Examining closely, we find 181 different stressmarks within the *Expert DSE* set that achieve the maximum core IPC. The minimum and the maximum power exhibited by them is 7% below and 9.6% above the baseline, respectively. These results depict how difficult it is to search for the optimal power stressmark. Even while achieving the same maximum IPC with the very same instruction types, the actual instruction sequence can affect the power consumption quite considerably.

The *MicroProbe* stressmark set, automatically defined using the functional unit, IPC and EPI information as heuristics, achieves similar results as the *Expert DSE*. In fact, it improves the max-power stressmark *Expert DSE* by approximately 1 percentage point. Also, this exceeds the maximum power observed during the execution of the entire SPEC CPU2006 suite by a 10.7%. These results confirm that EPI, IPC plus functional unit information provide good heuristics to constrain the DSE and systematize the max-power stressmark generation process without requiring expert knowledge.

Finally, the fact that systematically generated stressmarks slightly outperform the hand-crafted stress tests generated by an expert, confirms the utility of the proposed approach. Moreover, in a real measurement context, being able to constrain the search space to the actual points of interest is crucial in avoiding practical limitations posed by design space explosion.

7. Related work

Benchmarks and Micro-benchmarks: From the pioneering Whetstone [16] and Dhrystone [46] to current benchmark suites such as the SPEC CPU2006 [25], benchmarks are used for both academic research and comparative evaluation of existing solutions. Moreover, specifically designed benchmarks, named micro-benchmarks, are needed in several situations. For instance, they have been used to reverse engineer structure latencies [24] or branch organization [37,45], to evaluate performance, power or thermal efficiency [15,22,28,38] or to generate and calibrate models [8,9,12].

Micro-benchmark Generation Frameworks: The need of a systematic method to generate micro-benchmarks was identified back in the 1980's [47,48]. The number of frameworks proposed since then has been growing continuously corroborating their importance for the community. In contrast to our adaptive framework, particular solutions —without the micro-architecture semantics of MicroProbe— were developed for different purposes: to generate synthetic micro-benchmarks [2–4, 26], to be able to reproduce proprietary application behavior [30,32], to perform architecture explorations [31], to generate power or reliability stress tests [20,21,33,39], to evaluate energy efficiency of systems [13], or to model cache behavior [1].

Counter-Based Processor Power Models: Most of the previous work on counter-based power modeling uses top-down approaches to model processor power consumption [5, 10, 11, 23, 41]. As a result, they lose the level of decomposability provided by bottom-up approaches. Moreover, we only found the work of Jimenez *et al.* [29] proposing a top-down model for a SMT/CMP processor, the POWER6.

Regarding bottom-up modeling methods, Isci *et al.* [27] was the first to propose a heuristic-based bottom-up modeling method using as heuristic the area size of the functional units. Bertran *et al.* [7–9] then proposed a bottom-up modeling method, entirely based on micro-benchmarks. Nevertheless, none of these bottom-up methods modeled a CMP/SMT system such as the POWER7. Finally, Bircher *et al.* [10, 11] present a system-level bottom-up method to derive the power breakdown of the entire system (cpu, memory, disks, etc.).

Max-Power Stressmark Generation: The systematization of the generation of max-power stressmarks has been investigated for different environments. In [33], the authors present a micro-benchmark generation framework and show its utility for generating processor max-power stressmarks. In that work, the design space is defined by an abstract workload model. Then, genetic algorithms are used to find an optimal solution. Ganesan *et al.* [20] present a similar approach but targeting overall system power consumption, including processor and memory. The same authors extended the work to multi-cores [21] showing that when taking into account processor and memory power consumption, simple parallel execution of single core max-power stressmarks, do not exhibit the maximum power consumption. Our work in max-power stressmark case study is orthogonal to these works, since we focus on the importance of using micro-architecture semantics to constrain the search within the design space. We believe that these prior 'black-box' proposals are significantly improved by taking into account the extra information provided by MicroProbe.

8. Conclusion

In this paper, we present an adaptive micro-benchmark generation framework (MicroProbe), with three salient features that distinguish it from prior work: detailed knowledge of low-level micro-architecture semantics, flexible code generation support and integrated design space exploration support. To highlight these features of MicroProbe, we present experimental results centered around an IBM POWER7 CMP/SMT system. First, we produce a MicroProbe-driven empirical power model that estimates the power consumption of the SPEC CPU2006 benchmarks with average errors that are below 2.3%. Then, we conclude that micro-benchmark trained power models are more reliable across a broader range of contexts (normal and extreme power activities). We also use the framework to derive a taxonomy of POWER7 instructions based on energy-per-instruction (EPI). The characterization highlights the differences in energy consumption between instructions. Finally, we propose a method —based on EPI, IPC and functional unit information— to systematize the generation of power stress tests. The method is used to derive a stress test that exhibits a 10.7% increase in processor power over the maximum power seen during the execution of the SPEC CPU2006 benchmarks.

Acknowledgements

From the Barcelona Supercomputing Center (BSC) and the Universitat Politècnica de Catalunya (UPC) side, this work was supported by the Spanish Ministry of Education [Contracts #TIN-2007-60625 and #CSD2007-00050]; the Generalitat de Catalunya [Contract #2009-SGR-980]; the European Commission in the context of the HiPEAC Network of Excellence [Contracts EU FP7/ICT #217068 and #287759]; and the BSC-IBM collaboration agreement. From the International Business Machines (IBM) Corporation side, this work was supported by DARPA under the Computational Reliability Project [Contract #N66001-11-C-4027] and by the Lawrence Livermore National Laboratory (LLNS) under the BlueGene/Q Project [Subcontract #B554331]. Finally, we would like to thank Srilatha Manne and the anonymous reviewers for their comments and feedback.

References

[1] G. Balakrishnan *et al.*, "WEST: Cloning data cache behavior using stochastic traces," in *Proc. of HPCA'12*, pp. 1–12, Feb 2012.

[2] R. H. Bell Jr. *et al.*, "Efficient power analysis using synthetic testcases," in *Proc. of IISWC'05*, pp. 110–118, Oct 2005.

[3] R. H. Bell Jr. *et al.*, "Automatic testcase synthesis and performance model validation for high performance PowerPC processors," in *Proc. of IISWC'06*, pp. 154–165, Mar 2006.

[4] R. H. Bell Jr. *et al.*, "Improved automatic testcase synthesis for performance model validation," in *Proc. of ICS'05*, pp. 111–120, Jun 2005.

[5] F. Bellosa, "The benefits of event-driven energy accounting in power-sensitive systems," in *Proc. of EW'00*, pp. 37–42, Sep 2000.

[6] R. Bertran *et al.*, "POTRA: A framework for building power models for next generation multicore architectures," in *Proc. of SIGMETRICS'12*, pp. 427–428, Jun 2012.

[7] R. Bertran *et al.*, "Counter-based power modeling methods: Top-down vs bottom-up," *The Computer Journal*, vol. 99, pp. 1–16, Aug 2012.

[8] R. Bertran *et al.*, "A systematic methodology to generate decomposable and responsive power models for CMPs," *IEEE Trans. on Comp.*, vol. 99, pp. 1–14, Apr 2012.

[9] R. Bertran *et al.*, "Decomposable and responsive power models for multicore processors using performance counters," in *Proc. of ICS'10*, pp. 147–158, Jun 2010.

[10] W. Bircher *et al.*, "Complete system power estimation: A trickle-down approach based on performance events," in *Proc. of ISPASS'07*, pp. 158–168, Apr 2007.

[11] W. Bircher *et al.*, "Complete system power estimation using processor performance events," *IEEE Trans. on Comp.*, vol. 61, no. 4, pp. 563–577, Apr 2011.

[12] B. Black *et al.*, "Calibration of microprocessor performance models," *Computer*, vol. 31, no. 5, pp. 59–65, May 1998.

[13] K. D. Bois *et al.*, "SWEEP: Evaluating computer system energy efficiency using synthetic workloads," in *Proc. of HIPEAC'11*, pp. 159–166, Jan 2011.

[14] P. Bose *et al.*, "Bounds modelling and compiler optimizations for superscalar performance tuning," *J. Syst. Archit.*, vol. 45, no. 12–13, pp. 1111–1137, Jun 1999.

[15] D. Bull *et al.*, "A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation," in *Proc. of ISSCC'10*, pp. 284–285, Feb 2010.

[16] H. J. Curnow *et al.*, "A synthetic benchmark," *The Computer Journal*, vol. 19, no. 1, pp. 43–49, Feb 1976.

[17] S. Eranian, "Linux has a generic performance monitoring API!" in *Proc. of CSCADS'09*, p. 1, Jul 2009.

[18] L. V. Ertvelde *et al.*, "Benchmark synthesis for architecture and compiler exploration," in *Proc. of IISWC'10*, pp. 1–11, Dec 2010.

[19] M. Floyd *et al.*, "Adaptive energy-management features of the IBM POWER7 chip," *IBM J. Res. & Dev.*, vol. 55, no. 3, pp. 276–293, May 2011.

[20] K. Ganesan *et al.*, "SYstem-level Max POwer (SYMPO): A systematic approach for escalating system-level power consumption using synthetic benchmarks," in *Proc. of PACT'10*, pp. 19–28, Sep 2010.

[21] K. Ganesan *et al.*, "MAximum Multicore POwer (MAMPO): an automatic multithreaded synthetic power virus generation framework for multicore systems," in *Proc. of SC'11*, pp. 1–12, Nov 2011.

[22] G. Gerosa *et al.*, "A sub-2W low power IA processor for mobile internet devices in 45nm high-k metal gate CMOS," *J. of Solid-State Circ.*, 2009.

[23] B. Goel *et al.*, "Portable, scalable, per-core power estimation for intelligent resource management," in *Proc. of GREEN'10*, pp. 135–146, Aug 2010.

[24] J. Gonzalez-Dominguez *et al.*, "Servet: A benchmark suite for autotuning on multicore clusters," in *Proc. of IPDPS'10*, pp. 1–9, Apr 2010.

[25] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH News*, vol. 34, no. Sep, pp. 1–17, 4 2006.

[26] C. Hsieh *et al.*, "Microprocessor power estimation using profile-driven program synthesis," *IEEE Trans. on Comp.-Aided Design. of Integ. Cir. & Sys.*, vol. 17, no. 11, pp. 1080–1089, Nov 1998.

[27] C. Isci *et al.*, "Runtime power monitoring in high-end processors: methodology and empirical data," in *Proc. of MICRO'03*, pp. 96–108, Dec 2003.

[28] R. Iyer *et al.*, "Comparing the memory system performance of the HP V-class and SGI Origin 2000 multiprocessors using microbenchmarks and scientific applications," in *Proc. of ICS'99*, pp. 339–347, Jun 1999.

[29] V. Jiménez *et al.*, "Power and thermal characterization of POWER6 system," in *Proc. of PACT'10*, pp. 7–18, Sep 2010.

[30] A. Joshi *et al.*, "Performance cloning: A technique for disseminating proprietary applications as benchmarks," in *Proc. of IISWC'06*, pp. 105–115, Oct 2006.

[31] A. Joshi *et al.*, "The return of synthetic benchmarks," in *Proc. of SPEC Benchmark Workshop*, pp. 1–11, Jan 2008.

[32] A. Joshi *et al.*, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Trans. on Arch. & Code Opt.*, vol. 5, no. 2, pp. 1–33, Sep 2008.

[33] A. Joshi *et al.*, "Automated microprocessor stressmark generation," in *Proc. of HPCA'08*, pp. 229–239, Feb 2008.

[34] Y. Kim *et al.*, "Automated di/dt stressmark generation for microprocessor power delivery networks," in *Proc. of ISLPED'11*, pp. 253–258, Aug 2011.

[35] T. Li *et al.*, "Run-time modeling and estimation of operating system power consumption," pp. 160–171, Jun 2003.

[36] IBM staff, "Power ISA™. Version 2.06 Revision B," Jul 2010, [Online] http://www.power.org/resources/reading/.

[37] M. Milenkovic *et al.*, "Microbenchmarks for determining branch predictor organization," *Softw. Pract. Exper.*, vol. 34, no. 5, pp. 465–487, Apr 2004.

[38] S. Naffziger *et al.*, "The implementation of a 2-core, multi-threaded itanium family processor," *J. of Solid-State Circ.*, vol. 41, no. 1, pp. 197–209, Jan 2006.

[39] A. Nair *et al.*, "AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors," in *Proc. of MICRO'10*, pp. 125–136, Dec 2010.

[40] S. Polfliet *et al.*, "Automated full-system power characterization," *IEEE Micro*, vol. 31, no. 3, pp. 46–59, May 2011.

[41] K. Singh *et al.*, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH News*, vol. 37, no. 2, pp. 46–55, Jul 2008.

[42] B. Sinharoy *et al.*, "IBM POWER7 multicore server processor," *IBM J. Res. & Dev.*, vol. 55, no. 3, pp. 1–29, May 2011.

[43] D. C. Snowdon *et al.*, "Accurate on-line prediction of processor and memory energy usage under voltage scaling," in *Proc. of EMSOFT'07*, pp. 84–93, Oct 2007.

[44] V. Tiwari *et al.*, "Instruction level power analysis and optimization of software," in *Proc. of VLSI'96*, pp. 326–328, Jan 1996.

[45] V. Uzelac *et al.*, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Proc. of ISPASS'09*, pp. 207–217, Apr 2009.

[46] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Comm. of ACM*, vol. 27, no. 10, pp. 1013–1030, Oct 1984.

[47] W. S. Wong *et al.*, "Synthesizing benchmarks with appropriate instruction mix and locality," in *Proc. of ICCA'87*, pp. 1–12, Jun 1987.

[48] W. S. Wong *et al.*, "Benchmark Synthesis Using the LRU Cache Hit Function," *IEEE Trans. on Comp.*, vol. 37, no. 6, pp. 637–645, Jun 1988.

[49] W. Wu *et al.*, "A systematic method for functional unit power estimation in microprocessors," in *Proc. of DAC'06*, pp. 554–557, Jul 2006.