Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor

STEVE KERRISON and KERSTIN EDER, University of Bristol

This article examines a hardware multithreaded microprocessor and discusses the impact such an architecture has on existing software energy modeling techniques. A framework is constructed for analyzing the energy behavior of the XMOS XS1-L multithreaded processor and a variation on existing software energy models is proposed, based on analysis of collected energy data. It is shown that by combining execution statistics with sufficient data on the processor's thread activity and instruction execution costs, a multithreaded software energy model used with Instruction Set Simulation can yield an average error margin of less than 7%.

Categories and Subject Descriptors: C.1.4 [**Processor Architectures**]: Parallel Architectures; C.3 [**Special Purpose and Application-Based Systems**]: Real-time and Embedded Systems; C.4 [**Performance of Systems**]: Modeling Techniques

General Terms: Measurement, Performance

Additional Key Words and Phrases: Software energy modeling, multithreading, ISA-level energy modeling, embedded systems, computer architecture, XMOS XS1 xCORE

ACM Reference Format:
Steve Kerrison and Kerstin Eder. 2015. Energy modeling of software for a hardware multithreaded embedded microprocessor. ACM Trans. Embedd. Comput. Syst. 14, 3, Article 56 (April 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2700104

1. INTRODUCTION

The energy consumed by an embedded system is governed by the physical properties of the hardware components in use, the power management capabilities of that hardware, and the activity driven by the software running on it. It is the software that controls whether the system *needs* to be consuming energy and, if so, at what level of activity the underlying hardware needs to operate. Writing software that is a good fit to the underlying hardware has been identified as essential in the creation of energy-efficient and low-power systems [Roy and Johnson 1997].

Given the complexity of a modern embedded system, correlating software components with the energy consumed by the hardware is a complex process, understood by few. Yet, as we strive to reduce energy consumption for environmental, performance, or physical reasons, our understanding of the relationship between software and energy must improve. An embedded-software developer should have tools and knowledge at

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

Authors' addresses: S. Kerrison and K. Eder, Dept. of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom; emails: {steve.kerrison, kerstin.eder}@bristol.ac.uk.

DOI: http://dx.doi.org/10.1145/2700104

hone's disposal in order to have, at the very least, an intuition as to the energy that will be consumed by the software being written. Better still would be a detailed view of how the software's energy consumption is composed and where changes could be made to exploit the energy-saving capabilities of the target hardware.

The relationship between software and energy consumption on various processors has been researched, including x86 [Tiwari et al. 1994b] and DSPs [Sami et al. 2002; Ibrahim et al. 2008]. However, the recent move towards devices with many processing elements, such as System-on-Chip (SoC) and Network-on-Chip (NoC), adds complexity to the software/energy relationship that requires further research.

This article examines a modern hardware multithreaded embedded processor architecture, the XMOS XS1-L [May et al. 2008] "xCORE," to establish how such an architecture relates software behavior to energy consumption.

1.1. Contribution

We seek to further the understanding of energy modeling in hardware multithreaded and real-time embedded systems, and do so in the following ways.

First, the impact that hardware multithreading has on the software/energy relationship is examined. Most interestingly, it is shown that architectural decisions that make hardware multithreading easy to understand and use from both a hardware design and software function perspective make the energy analysis of the XS1-L more complex than that of its single-threaded counterparts.

Next, a measurement framework for profiling the energy behavior of the XS1-L processor is developed. Data from this is then used to build on existing approaches to modeling software energy, with proposed adaptations to them accounting for the complexities introduced by hardware multithreading. An initial model is produced using a subset of the XS1 instruction set.

A series of benchmarks are run in order to evaluate the initial model. The measurement framework and initial model provide insight into what are the most significant factors in the software implementation that impact the energy consumption of the processor.

Finally, future research is detailed that can both extend the scope and improve the accuracy of the model, utilizing static analysis of code, linear regression techniques, and by modeling the I/O voltage domain to account for communication and device interface costs.

1.2. Conventions

In this article, the terms power and energy are used frequently. These terms are often interchanged in literature, but in the context of this article it would not be appropriate to do so. For clarity, therefore, their definitions are given.

Power, P, or *power dissipation*, is an instantaneous measure of a *rate of energy transfer*, or the rate at which work is done. It is quantified in watts, or W. Energy, E, or *energy consumption*, is a measure of *total work done*—the amount of potential that is transformed in order to achieve the desired outcome. In the case of a microelectronic system, this is the amount of electrical charge that is transferred during operation, much of which is transformed into heat in the system's various components. Energy is the integral of power over a period of time, $E = \int_0^T P(t)dt$. It is typically expressed in joules, or J. Applying both the concepts of energy and power, a system that sustains a power dissipation of 1W for 1s will have transferred 1J of energy.

It is important to note that reducing power and reducing energy are not necessarily equivalent goals. For example, it may be desirable to operate a processor at a higher power level to allow a task to be completed faster, then power down the processor, thus

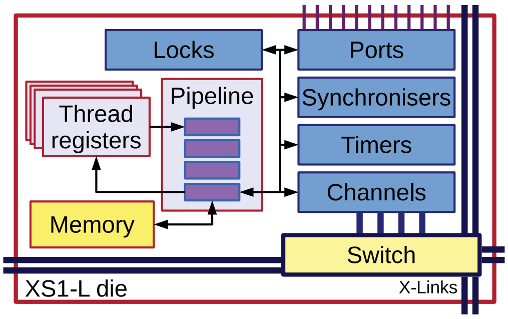


Fig. 1. Block diagram of XS1-L processor.

reducing the overall energy consumed. Conversely, there may be an ample supply of energy, but thermal limitations may limit how much power a system can dissipate.

2. ANATOMY OF A HARDWARE MULTITHREADED EMBEDDED PROCESSOR: THE XMOS XS1-L

The XMOS XS1-L processor, depicted at a block level in Figure 1, is an embedded processor that allows hardware interfaces to be written in software. The execution of software is time-deterministic and the instruction set places I/O hardware extremely close to the software. This provides timing guarantees and advanced GPIO capabilities that mean that components such as UARTs, MII Ethernet, SPI, I2C, and USB can be expressed as software rather than fixed hardware units. The application space of this processor is therefore between a traditional programmable microprocessor and an FPGA.

Energy efficiency is sought both in the hardware and programming model by introducing *event-driven* paradigms, whereby a thread may defer its own execution until the hardware observes a particular event, such as a change in state of an I/O pin. The hardware scheduler eliminates the need for polling loops and similarly, due to multithreading, interrupt service routines are not required.

The architecture also features a custom asynchronous interconnect that can be used to assemble networks of XS1-L processors to distribute programs over many cores, with communication taking place over 2- or 5-wire “X-Links.” This article focuses on multithreaded operation of a single core, although the network capabilities will be used for test synchronization. Examination of multicore XS1-L systems is a goal for future research.

2.1. General Multithreading Methods

At the center of the XS1 architecture is a multithreaded pipeline. To provide clarity in the details of this particular multithreaded implementation, we briefly review the various ways in which multithreading can be achieved.

A *thread* is a sequence of execution in a single context. A multithreaded system can switch between these contexts, using hardware and/or software mechanisms. Threads are sometimes also termed tasks or processes, although these are typically higher-level terms for activities encompassing one or more threads.

Multithreading can be achieved through the Operating System (OS), where the kernel implements a scheduling method depending on the purpose of the OS [Bovet and Cesati 2005; Microsoft Corporation 2012]. There are portable interfaces for multithreaded software, such as those defined in POSIX [Butenhof 1997].

In hardware, multithreading can be achieved with multiple decode units serving a group of execution units, such as Intel Hyper-Threading [Intel Corporation 2003], which provides two front ends per core, or the Sun Sparc Niagara [Kongetira et al. 2005], which has 8 cores and 4-way multithreading per core. Multiple cores are now also common across the computing spectrum, including multicore CPUs with 4 or more cores per package and GPU accelerators capable of executing dozens or hundreds of threads concurrently.

2.2. The XS1-L Multithreaded Pipeline

To bring the software as close to the physical interfaces as possible, the XS1 architecture manages threads in hardware, with machine instructions and hardware resources dedicated to thread creation, synchronization, and destruction [May 2009]. In addition, each thread has its own bank of registers, containing a program counter, general purpose and special purpose registers. This removes the overhead of an OS and allows threads to be created in as few as 3 instructions, but places a hard limit on the number of threads that can exist on the processor at any one time. In the case of the XS1-L, up to 8 threads are supported.

These 8 threads are executed in a round-robin fashion through a 4-stage pipeline. The pipeline avoids data hazards and dependencies by allowing only a single instruction per thread to be active within the pipeline at any one time. As such, the pipeline is only full if 4 or more threads are active. If there are less than 4 threads able to run, then there will be clock cycles in which pipeline stages are inactive. This makes the microarchitecture relatively simple to reason about, but requires at least 4 active threads in order to use the full computational power of the processor. When more than 4 threads are active, maximum pipeline throughput is maintained, but compute time is divided between the active threads.

2.2.1. Calculating Instruction Throughput. With a 4-stage pipeline structure, the Instructions computed Per Second by the processor core, IPS_c, in relation to the core frequency, F, is simply $\mathrm{IPS}_p = F$, for the case in which the pipeline is full. It can be expressed for any number of threads, N_t, as shown in Equation (1).

$$\mathrm{IPS}_p = F\frac{\min(4, N_t)}{4}. \tag{1}$$

The instruction frequency of an individual thread, IPS_t, can similarly be expressed as:

$$\mathrm{IPS}_t = \frac{F}{\max(4, N_t)}. \tag{2}$$

An XS1-L processor typically operates at 400 or 500MHz, giving a maximum instruction throughput of 400MIPS, or 100MIPS per thread at 400MHz, and 500MIPS for the core with 125MIPS per thread at 500MHz.

2.2.2. Context Switching Free in Time, but not Energy. By virtue of the XS1's hardware threads and 4-stage pipeline, the processor effectively performs a "context switch" on every clock cycle, assuming there is another thread in an active state. In terms of performance, this delivers significant benefits to multithreaded programming. From an energy perspective, however, this creates interesting behavior.

First, with each clock cycle comes a completely new set of states into the pipeline stage—the proceeding instruction and data will be from the next thread in the schedule. This introduces an energy cost through switching in the control and data logic of the processor. Second, these context switches on every cycle change the energy characteristics of the pipeline in a way that existing models cannot account for. This is examined further in Section 3.

3. BUILDING A MODEL OF A MULTITHREADED PIPELINE

In this section, various existing software energy modeling approaches are considered. These approaches tend to model single-threaded architectures, or architectures with parallel execution that is achieved using a different method than hardware multithreading. Such approaches can be considered against the behavior of the XMOS XS1-L pipeline, as a basis for building an appropriate model for this architecture.

3.1. Existing Single-Threaded Modeling Methods

Tiwari's early work into x86 instruction set modeling [Tiwari et al. 1994b] seeks to estimate the energy, E, of a program, p, by considering three components of execution: the *base cost* of each instruction that is executed, the *interinstruction overheads* of switching between one instruction and another, and any *external effects* such as cache misses. These values are extracted from a target system with a test harness executing specific instruction sequences and measurement equipment collecting energy consumption data. The model is expressed by Equation (3) [Tiwari et al. 1996]. For all instructions i in the target ISA, the base cost B_i is multiplied by N_i the number of times the instruction i is executed. For each pair of instructions executed in sequence i, j, the interinstruction overhead $O_{i,j}$ and frequency of occurrence $N_{i,j}$ is counted. Finally, for each external component k, the energy cost of external effects E_k is determined, for example, with an external cache model.

$$E_p = \sum_i (B_i \times N_i) + \sum_{i,j} \left(O_{i,j} \times N_{i,j}\right) + \sum_k E_k \tag{3}$$

Building on this research are energy modeling tools such as the Wattch framework [Brooks et al. 2000]. Wattch produces energy estimates of software through simulation by modeling key components of a processor architecture, such as the cache hierarchy and size, functional unit utilization, and branch prediction capabilities. Wattch can model software targeting various architectures to within 10% of commercial low-level hardware modeling tools. The SimpleScalar [Austin 2002] architecture-modeling software was used as a basis for a similar power model, resulting in Sim-Panalyzer [Panalyzer 2004]. The idea of measuring instructions and their interactions can be further decomposed, for which a model was proposed by Steinke et al. [2001]. This extracted more information on the source of energy consumption in the processor pipeline, such as the cost of switching in each read action on the register file, as well as the cost of addressing different registers for read and write-back. The precision of the approach was shown to be within 1.7% of the target hardware, although it significantly increases the number of variables that must be considered when implementing the model. Other types of processor architectures have also been modeled in similar ways, such as VLIW DSPs [Sami et al. 2002; Ibrahim et al. 2008], with average accuracies of 4.8% and 1.05%, respectively.

Alternative approaches exist that aim to deliver similar functionality through modeling at a different level, from block-level modeling, of which processor components are exercised by code [Ibrahim et al. 2008], to the creation of databanks that characterize the energy behavior of the most frequently used software library calls in a particular system [Qu et al. 2000], with a 3% average error.

Performance counters can be mapped to the power dissipation of a processor and its memory hierarchy. A model has been demonstrated for the Intel PXA255 XScale (ARMv5-based) processor [Contreras and Martonosi 2005], with estimated average power dissipation within 4% of the measured average. *Power weight* terms in the model can be adjusted to accommodate changes to voltage and frequency. There are no hardware performance counters documented for the XS1-L, excluding this particular approach as a basis for our research.

Similar approaches can be applied in coprocessors used in high-performance computing, such as the Xeon Phi. An energy model of the Xeon Phi [Shao and Brooks 2013] provides energy per instruction data based on characterization from micro benchmarks and hardware performance counters. The model was used to identify redundant prefetch operations that were wasting energy. The Phi has a multilayer cache network, making it significantly different from the memory in the XS1-L.

To model complex micro-architectures, linear regression analysis can be used. With sufficient supporting empirical data, a solution to a parameterized model can be found that establishes values for any unknown terms. This has been utilized to model an ARM7TDMI processor [Lee et al. 2001], using empirical energy data from observed test programs to aid the solver, yielding a model with a 2.5% average error.

These approaches can all deliver an accuracy of 1% to 10% across various architectures. However, the architectures that they analyze are either single-threaded, special purpose DSPs, or are not targeted at embedded real-time systems. As such, these models are not equipped to capture a hardware multithreaded embedded processor. This article proposes methods that extend some of these techniques by supporting such devices, with the aim of achieving comparable accuracy to prior models.

3.2. Examining the XS1-L Architecture

The relatively small instruction set and deterministic execution of the XS1-L processor would appear to lend itself to a model similar to that created by Tiwari et al. [1996], as described in the previous subsection and Equation (3). Therefore, this approach is taken as a starting point.

Such a model requires a *base cost* for instructions as well as *interinstruction overheads*, plus any other effects not directly expressed by the stream of instructions that are executing. To construct a model in such a style, power measurements must be taken for individual instructions as well as measurements for pairs of instructions, so that the instruction base cost and interinstruction overhead can be considered. This data can then be used to estimate a program's energy, based on the sequence of instructions that it will run.

3.2.1. Base Costs. The base cost defines a minimum energy consumption that is always present during the execution of a program. In a sequential, single-threaded microprocessor, a no-op instruction could represent the energy consumed when the processor is idle. The costs of executing meaningful instructions and the interactions between them can then be built on top of this base cost.

Taking into account the XS1-L's event-driven architecture and therefore idle times in which no instructions execute, the base cost can be defined as the minimum energy consumed when there are no active threads. Investigation into and establishment of a base cost for this processor is detailed in full in Section 4.

3.2.2. Interinstruction and Interthread Overheads. Once a base cost is established, the next challenge is how to handle instructions and interinstruction overheads in the context of the XS1-L.

In order to establish these in a similar fashion to the Tiwari et al. [1996] model, the cost of executing each instruction and of transitioning between pairs of instructions must be determined. Hardware measurements are required in order to establish the magnitude and variability of interinstruction overheads so that an appropriate granularity can be chosen for the model, delivering an acceptable performance/accuracy trade-off. For example, if the contribution of interinstruction overhead is insignificant in comparison to the cost of each individual instruction, then it may not be necessary

Table I. Comparison of Key Differences Between Various Architectures

Feature	XS1-L	ARM7TDMI [Lee et al. 2001]	C641T [Ibrahim et al. 2008]	Xeon Phi [Shao and Brooks 2013]
Cores	1	1	1	60+
Threads	8	1	1	4
Instr. sched.	Round-robin threads, in order	In-order	In-order 2x4 VLIW	In-order
Forwarding	No	Yes	No	Yes
Com. model	Channels	Shared memory		
Mem./cache	1-cycle SRAM, no cache	Optional caches	L2	L2 + tag-cache on ring network

to consider it, or it may be appropriate to generalize it if there is little variation in overhead between instructions.

To produce a model appropriate for multithreaded hardware, the processor must be seen as a pipeline that is executing a stream of unrelated instructions from concurrently executing threads. Although dependencies and synchronization may exist between *some* threads at a higher level of abstraction, on a per-instruction level, a pair of instructions that progress together sequentially through the pipeline are effectively unrelated in any real-world embedded application.

This precludes using a sequence of instructions in a thread as a means of exercising the processor in order to determine instruction costs and interinstruction overheads. Instead, instruction overheads must be measured by controlling the instructions that a *collection of threads* are running, such that the exact sequence of instructions passing through the processor pipeline is known. Section 4.3 describes how the measurement framework achieves these guarantees in order to extract interinstruction overheads.

3.3. Simulation Performance

For a software energy model to be useful, it must be more convenient to run it than to instrument and measure a hardware system. The modeling approach used in this article requires the use of an Instruction Set Simulator (ISS). On a 2.26GHz Intel Core i3 CPU, a full instruction trace simulation using the standard XMOS tool `xsim`[1] takes 51 minutes for a 0.4s real-time benchmark. A simulation producing only execution statistics, using the faster `axe`[2] tool, takes 40s.

Thus, there is motivation to construct a model that can rely on instruction statistics rather than complete trace data. However, statistics alone make it impossible to account for interinstruction overheads at a per-instruction level. The impact of forgoing this must be considered during data collection.

3.4. Architecture Comparison

Table I illustrates the key differences between the target processor for this research and a sample of other processors used in previous work described in Section 3.1. The significant differences in pipeline implementation, threading methods, communication model, and memory hierarchy serve to justify the goal of this work in creating the foundations of a model for the XS1-L.

4. XMPROFILE: A FRAMEWORK FOR PROFILING THE XS1-L

In order to acquire data to enable the creation of a software energy model for the XS1-L, a hardware–software measurement framework was constructed, as shown in Figure 2.

[1]http://www.xmos.com/resources/downloads.
[2]https://github.com/stevekerrison/tool_axe.

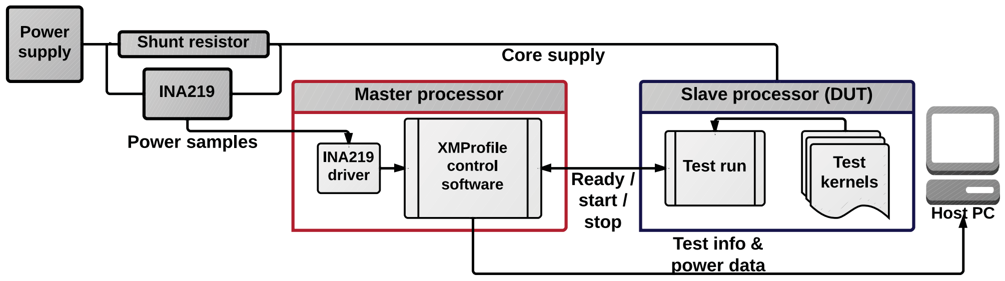


Fig. 2. Overview of test harness hardware and software structure.

It has the following aims:

(1) To execute code with a level of granularity that delivers certainty as to the trace of instructions through the pipeline
(2) To provide a measurement interface in order to easily collect energy data and attribute it to test cases
(3) To perform constrained generation of tests for automation of the profiling process
(4) To support the inclusion of benchmark code to enable comparisons between the resulting model and the actual energy characteristics of the target hardware.

4.1. Hardware

The hardware platform for the energy-profiling effort consists of 2 XS1-L devices, an XK-1 development board containing the *master* processor and a bespoke XMOS board containing an additional XS1-L—the *slave* processor or the Device Under Test (DUT). The bespoke board was modified to provide easy access to the core power supply of its XS1-L. The XK-1 development board controls a DC-DC power supply and an INA219 power measurement chip,[3] allowing power dissipation of the power supply to be sampled at a rate of up to 8K samples per second with up to 11-bit resolution with a least-significant sample bit of 680μW for the expected maximum current of the XS1-L.

In addition to controlling and monitoring the power supply of the DUT, the master processor is also responsible for synchronizing tests against energy measurements in order to automate the collection of model data.

4.2. Software

The `XMProfile` suite of software is composed of four key components:

(1) Power measurement and data streaming to host PC
(2) Test loading and synchronization with power measurements
(3) Test case generation with constrained random data for all instruction permutations in a given instruction subset
(4) Test control software, delivering fine-grained management of instruction flow during test kernel execution

The master processor runs software that samples power values from the INA219. These samples are then streamed out over a USB interface to a host PC. At the end of each test run, an average power figure is calculated. This combination of streamed data and test-run averages provides sufficient data to construct an energy-consumption model.

[3]http://www.ti.com/product/ina219.

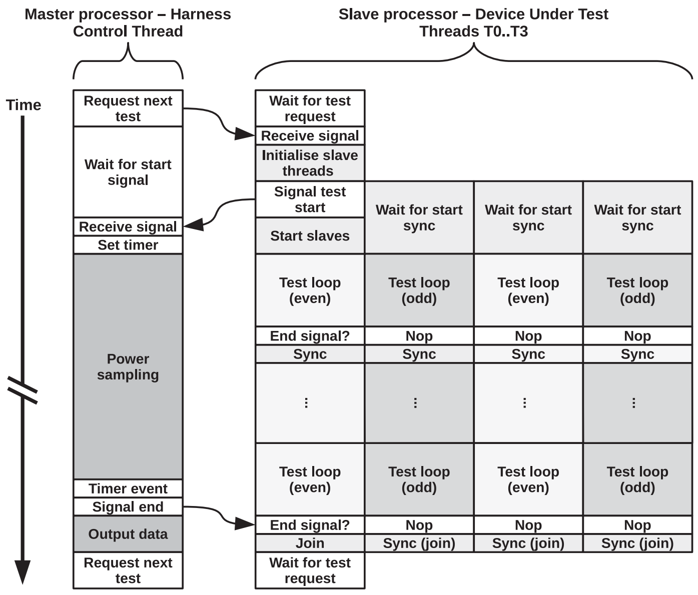


Fig. 3. Test harness and DUT process flow.

Tests are synchronized with power measurements by using the XMOS XS1's communication architecture. The master processor and DUT form a network over a 2-wire X-Link. The link is used as a trigger to signal the start of the next test and halt the test once the test period is over.

4.3. Controlling the Pipeline

Establishing the instruction costs and overheads through the pipeline requires the ability to control the order of instructions progressing through it. When subsequent instructions come from different threads, this is difficult to guarantee at a high level, such as with compiled C code. However, the XS1-L's single-cycle thread synchronization allows the test harness to have precise control over which instructions in a thread are executing at any one time, provided that the tests do not introduce any nondeterminism with respect to execution time, such as through I/O operations. A typical test flow is depicted in Figure 3.

A test thread is a loop containing the body of instructions to be profiled, with minimal prologue and epilogue, but sufficient to ensure synchronization and allow correct termination. Four threads, T0 to T3, are required to fill the pipeline and create instruction interactions on every clock cycle. To observe interinstruction effects, the body of odd-numbered threads are populated with one instruction I_{odd}, while even-numbered threads are populated with another I_{even}. As the threads execute round-robin, the instruction executed at a given pipeline stage will alternate between I_{odd} and I_{even}, allowing specific interinstruction effects to be measured.

All threads are synchronized against the thread that creates them—the master thread. In this case, T0 is the master and T1 to T3 are its slaves. As such, the loop prologue and epilogue of the master thread is slightly different to that of the slave threads. During the test, at the start of each loop the slaves perform a synchronization (`SSYNC` instruction) against the master thread (`MSYNC` instruction). If the master has received the end-of-test signal from the test harness, then it performs an `MJOIN` instead of an `MSYNC`. This kills the slave threads when they next synchronize.

Table II. Representation of Instruction Sequence for Various *Active* Thread Counts

Timestep	1 thread	2 threads	3 threads	4 threads	5 threads
1	$T_{0,0}$	$T_{0,0}$	$T_{0,0}$	$T_{0,0}$	$T_{0,0}$
2	—	—	$T_{1,0}$	$T_{1,0}$	$T_{1,0}$
3	—	$T_{1,0}$	$T_{2,0}$	$T_{2,0}$	$T_{2,0}$
4	—	—	—	$T_{3,0}$	$T_{3,0}$
5	$T_{0,1}$	$T_{0,1}$	$T_{0,1}$	$T_{0,1}$	$T_{4,0}$
6	—	—	$T_{1,1}$	$T_{1,1}$	$T_{0,1}$

Threads are represented as $T_{n,i}$, for thread number n and instruction number i.

To minimize the overhead of the execution of loop prologues and epilogues, the loop body must be sufficiently long. The number of body instructions N_b required to achieve a body-to-total instruction ratio R with overhead instructions N_o is determined through Equation (4). This work specifies $R = 0.95$ and $N_o = 4$.

$$N_b = \left\lceil \frac{N_o R}{1 - R} \right\rceil \tag{4}$$

Thread schedule. Initial testing yielded an interesting discovery in relation to how threads are scheduled into the pipeline. With a single active thread, an instruction is issued once every 4 clock cycles. When two threads are active, instructions are issued every other clock cycle. The alternative would be to issue 2 instructions (one from each thread), and then have 2 cycles for which no instructions are issued. This is functionally equivalent, but it may have energy implications because it affects the switching within the pipeline.

For 3 threads, an instruction is issued for 3 in every 4 clock cycles, and for 4 or more active threads, an instruction is issued on every clock. Allocated, but inactive threads (i.e., threads waiting on events) do not issue instructions, thus have no influence on scheduling. Table II illustrates how instructions are issued.

4.4. Generating Tests

Blocks of instructions are required to fill the loop bodies of test threads, the expectation being that the majority of test time will be spent executing those body instructions, giving a power figure for those instructions. A number of ALU instructions were hand-coded into test loops to help gain understanding of what to expect and to determine a good approach for automation.

Following this initial setup, the process of creating tests was largely automated. For 36 arithmetic operations, a test was generated for every possible pairing of them.

To account for data variation, constrained random data as well as constrained random source and destination operands were generated. The constraints applied ensure that for each instruction the supplied data are valid (i.e., cannot cause an exception condition) and that results do not overwrite source registers, avoiding value convergence over the course of the loop body.

Constrained random data generation was also used to provide different data widths to the test loops, with bit-widths of 32 (full width), 24, 16, 8, 4, 2, 1 and 0. Bit-masking was applied at code generation time to constrain the data range, thus the test loops themselves were identical between runs at various data widths.

Thus far this approach has been applied to 36 of the 179 instructions, principally covering arithmetic operations in the CPU. This excludes branches, I/O, communication, and other resource-related instructions. These other instructions can affect control flow or have nondeterministic timing. Although it is possible to build test loops that utilize them, it has been necessary to start with pairs of arithmetic operations so that the data from these can be used in combination with data from loops that use more complex sequences of instructions. This allows the costs of the other instructions to be extrapolated and is discussed further in Section 7.

The process to generate tests for each ALU instruction automatically is as follows:

(1) Describe constraints on all immediate encodings (value range or set of possible values).
(2) Describe characteristics of each instruction in terms of length, encoding, operand count (source and destination), immediate type, and the number of source/destination registers to allocate.
(3) For each unique pairing of instructions, generate odd and even threads for a test kernel, with the following generated contents:
   —For each instruction in a test, generate random starting values for the source registers within that instruction's constraints.
   —For each instruction in a test, generate random source and destination registers within specified range.
   —If an instruction has an immediate value, generate a random immediate within constraints.
   —Generate N_b instructions, satisfying Equation (4).
(4) Add test to list of tests to run.
(5) Compile group of tests into framework, split into separate binaries if the processor's program memory limit is exceeded.

As an example of a set of constraints, take the instruction `ashr`, *arithmetic shift right*, with an immediate shift value. In C, `Y = X >> I`, for a `signed int`, `X`, and a constant shift amount, `I`, is functionally equivalent to `ashr`. This is encoded as a 32-bit instruction. It has one input register and one output register, with an immediate value $I \in \{1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32\}$, encoded together with the register addresses into 11 bits. The operands of the instruction are constrained by these parameters to ensure that only valid assembly instruction sequences are generated.

With this process in place, energy data for the majority of arithmetic instructions can be collected in a matter of hours. Discussion and analysis of these results is presented in the following section.

5. ANALYSIS OF RESULTS

The test framework and the collected energy data yielded four key areas of results:

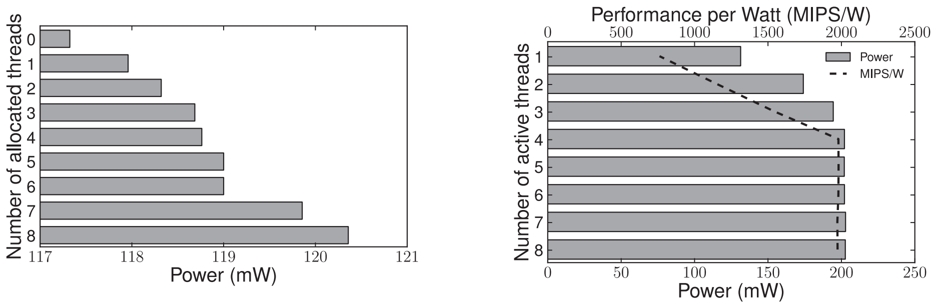
—Identification of the costs for executing a variable number of threads
—The energy consumed by different arithmetic instruction types
—The impact of interinstruction overheads in concurrent threads
—Demonstration of the impact of data values on processor energy consumption

These areas are each explored in detail within this section.

5.1. Base Cost

Investigating the base cost, as discussed in Section 3.2.1, it is observed that when a thread is waiting on an event, such as communication from another thread, an I/O event, or a timer comparison, it is descheduled and no more instructions from that thread are executed until the occurrence of one of the events it is waiting for. However, experiments show that the number of threads allocated, even if they are all descheduled and waiting for events, has an impact on system energy that can be attributed to the activity in the thread scheduler of the processor.

If the XS1-L's software energy model needs to consider not just the instructions that a thread is executing but the number of threads that are executing, then the base cost should also aim to capture the energy used when no threads are allocated.



(a) Base costs analysis with various allocated but otherwise idle threads.

(b) Thread costs for add instructions on random data with performance per watt overlaid.

Fig. 4. Active and inactive thread costs for the XS1-L processor.

This scenario was created by constructing a program that contained only a single thread that was subsequently released via the FREET (free current thread [May 2009]) instruction, leaving no allocated threads. Indeed, this yielded the lowest observed energy consumption when compared to any number of threads that were idle but still allocated, as shown in Figure 4(a).

This data gives a base cost figure that is independent of both instruction sequences and the number of active threads, creating a stable *minimum energy value* on which the rest of the processor energy model can be built, wherein active processor behavior can be considered.

There is a nonlinear relationship between the number of allocated threads, their state (active or waiting), and the energy consumption of the processor. An allocated thread when idle adds approximately 342μW to the processor's power dissipation. This is lower than the least significant bit of the power sampling hardware, thus is subject to the effects of both noise and data averaging.

The idle power of the XS1-L processor may be considered relatively high. However, this is operating at 400MHz, 1 V, in a 65nm process technology and with no power gating. Even with all threads idle, port logic, clock trees, PLLs, the scheduler, and a network switch are still active, such that I/O event response can begin at full speed within 10nS. Power-saving features such as a lower voltage, or a deep sleep with external wake-up, can be implemented with additional peripheral components, and frequency scaling is natively supported [May et al. 2008].

5.2. Thread Cost

As the number of *active* threads increases, so too does power dissipation, although the increase is less significant above 4 threads as the pipeline is always full beyond this thread count. Figure 4(b) demonstrates this behavior. The step in energy between 1 and 2 threads is greater than the step between 2 and 3 threads. This is believed to be related to the way in which smaller numbers of active threads are scheduled, as discussed in Section 4.3. When 2 threads are scheduled, the pipeline transitions between active and idle twice as frequently as with 1 or 3 active threads. The performance per watt of the processor at each thread count is overlaid onto Figure 4(b), as per IPS_c in Equation (1), highlighting the inefficiency of running less than 4 threads.

This characteristic bears similarities to the behavior of the Xeon Phi [Shao and Brooks 2013], for which instruction issue restrictions in the pipeline limit the energy efficiency of single-threaded execution on a core. However, the characterization of the two processors deviates significantly at the memory hierarchy and communication model, particularly when considering a larger system-level view, which is discussed in Section 7.

From this data, a baseline figure for the energy consumption of threads can be established, which can then be used as a component of the model to account for the number of allocated and active threads observed during simulation. The operations performed by the active threads must also be considered and combined with these baselines to account for thread activity.

In summary, both the cost of allocating a thread and the energy characteristics of various numbers of threads have been profiled. The respective data form an integral part of the multithreaded software energy model.

5.3. Instruction Cost

The cost of individual instructions and the interinstruction overheads are closely connected, thus they are considered simultaneously.

Using an approach similar to Tiwari et al. [1994a], the measured power for a given pair of instructions $m_{i,j}$ is represented in array M. The average power between them is calculated to give an estimate of the interinstruction overhead as array E, where $e_{i,j} = \frac{m_{i,i}+m_{j,j}}{2}$. Then the actual overhead, or the difference between estimated and measured power, $A = M - E$, is calculated.

Figure 5 is a depiction of these arrays for 32-bit constrained random data. Data are represented as “heat maps,” in which the color represents the measured power. The axes of the graphs show instructions together with their encoding. For example, `add_3r` is an `add` instruction with 3 operands (two source registers and one destination register), as defined in the XS1 architecture manual [May 2009]. A brief summary of the encodings is presented in Table III. Axes in the graphs are grouped by instruction operand count and separated by dashed lines, then sorted along the diagonal by individual instruction power in each group: that is, the power observed when all threads are executing the same instruction, thus there is minimal interinstruction overhead in the pipeline.

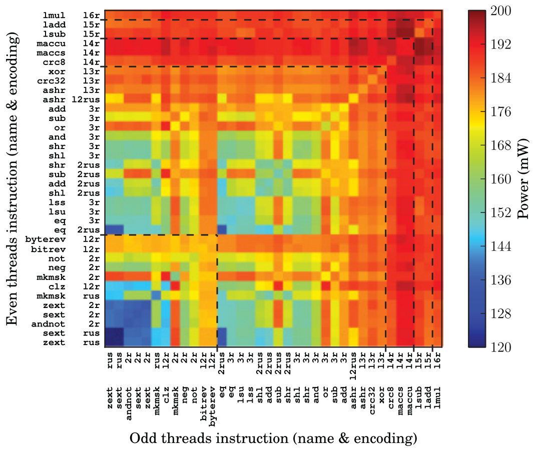
Each point on the grid of the heat map is a measurement of the power taken during interleaved execution of the instructions indicated by the axes. The color map is scaled to encapsulate the maximum and minimum observed values during the test suite and applied to the measured and estimated values for consistency and ease of comparison. The third map is independently scaled in order to expose where the overhead is significantly different from the baseline estimate.

Figure 5(a) is the measured power M. The diagonal of this map shows the instruction base cost, when there is no interinstruction effect. The lower triangle represents the power from interleaving every instruction in the test set and is mirrored into the upper triangle.

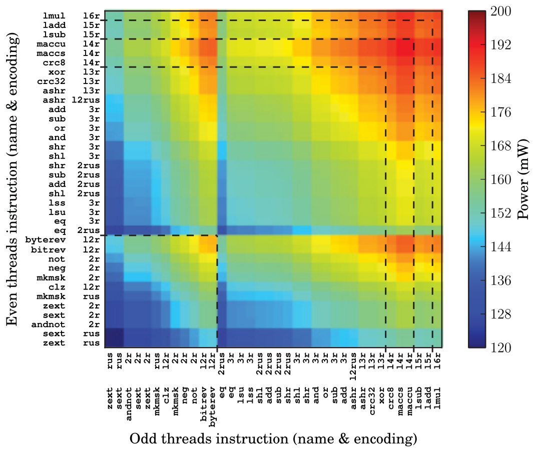
From the measured costs in the diagonal of Figure 5(a), the estimated interinstruction power E is calculated and shown in Figure 5(b). The actual overhead A is then shown in Figure 5(c).

Inspecting the axes groupings and instruction ordering shows that instructions that use a larger number of operands (`l4r`, `l5r`, `l6r`) are typically towards the top-right of the map because they dissipate more power. In addition to using more operands, they are encoded as long instructions, occupying 32-bits per instruction rather than 16-bits thus instruction fetches must be performed twice as often in order to carry them out. In the 2 to 3 operand regions, there is a greater number of instructions and a greater variation in power. The maximum variation in instruction power is as much as 40% of the total core power.

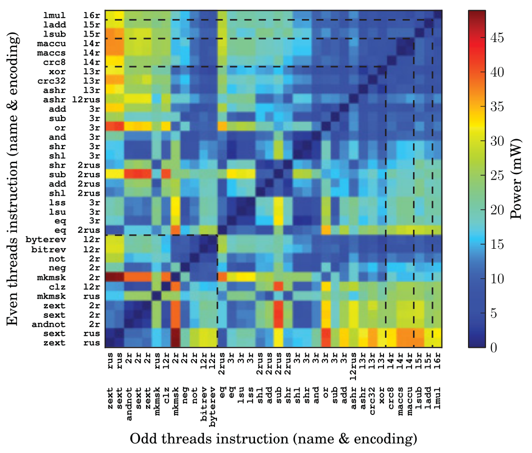
The difference between the actual interinstruction overheads and the estimation based on averaging is typically less than 10 mW. This is an order of magnitude lower than the instruction power. With such a small impact on power, precise calculation may not be necessary in order to produce an effective model.



(a) Measured power, M



(b) Estimated interaction, E

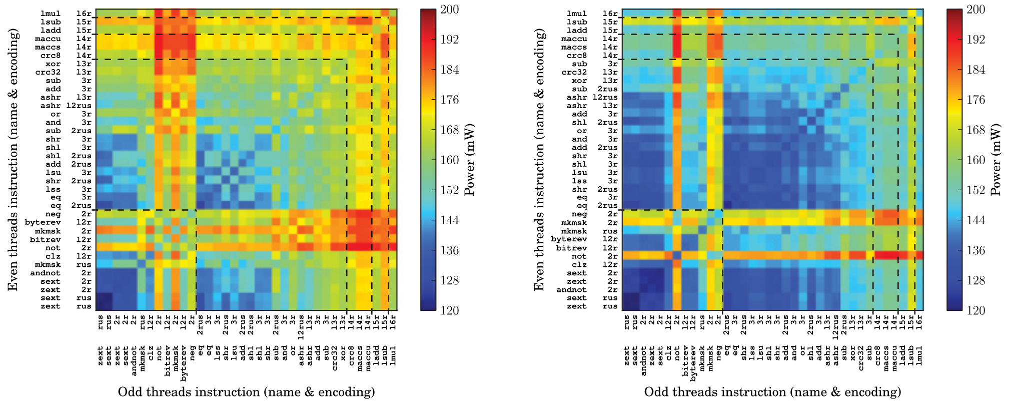


(c) Inter-instruction overhead, A

Fig. 5. Instruction power data and interinstruction overhead calculation for 32-bit data; dashed lines indicate a change in operand count.

Table III. Instruction Encoding Summary for the XS1 Instructions Under Test

Encoding	Source registers	Destination registers	Immediate	Instruction length (bits)
`rus`	1	1*	1	16
`2r`	2	1*	—	16
`l2r`	2	1*	—	32
`2rus`	1	1	1	16
`3r`	2	1	—	16
`l2rus`	1	1	1	32
`l4r`	3	1	—	32
`l5r`	3	2	—	32
`l6r`	4	2	—	32
*Destination operand address is the same as first source operand.				



(a) 16-bit data (b) 8-bit data

Fig. 6. Instruction power data for various data widths, with dashed lines denoting a change in operand count. A reduction in power can be seen for narrower data.

With these data and analysis, the instruction power, combined with the observed interinstruction overheads, can be incorporated into the software energy model. In addition to this, simplification of the model may be possible if the average power per operand count yields sufficient accuracy. However, the variation in instruction cost is significant enough that it is not appropriate to consider all instructions equal in this particular case, unlike in some other architectures, such as that examined by Russell and Jacome [1998]. Finally, due to its limited impact, the interinstruction overhead does not necessarily need to be considered for every possible instruction combination.

5.4. Data Width's Impact on Processor Energy Consumption

Given the observation in Section 5.3 that the number of operands has a significant affect on the power, it was hypothesized that a significant relationship exists between data values and processor energy consumption. The Tiwari model does not account for data, although variations on it do, such as the model of Steinke et al. [2001]. Therefore, it was necessary to investigate the significance of data width's impact on the XS1-L's energy consumption.

Test runs were re-executed for several constrained random datasets, in this case 0, 1, 2, 4, 8, 16, and 24 bits in addition to the 32-bit data already collected, as described in Section 4.4. Using the same plotting technique as described in Section 5.3, Figure 6 shows measured power data M for data widths of 16 and 8 bits.

Table IV. Interleaved `lmul` Calculations and Hamming Weight of Inputs and Outputs during Thread Transitions

	Input	Output
T0 and T2	`0x55555555,0x55555555,0x55555555,0x55555555`	`0x1c71c721,0xe38e38e3`
T1 and T3	`0xaaaaaaaa,0xaaaaaaaa,0xaaaaaaaa,0xaaaaaaaa`	`0x71c71c72,0x38e38e38`
XOR	`0xffffffff,0xffffffff,0xffffffff,0xffffffff`	`0x6db6db53,0xdb6db6db`
Hamming weight	128	42

Table V. Power Measurements for `lmul` Under Differing Data Conditions

0-bit data	16-bit data	32-bit data	Worst case data
131mW	164mW	189mW	222mW

Figure 6 shows that as we restrict the data width, the energy consumed by instructions drops, with a few exceptions. The extent of the reduction is dependent on the operation being performed. For example, an addition operation will at most produce a number 1-bit longer than its longest source operand, whereas a multiplication may produce a number twice as long (assuming no truncation due to overflow). Exceptional cases such as `mkmsk` (make mask, for producing bit masks) and `not` (bit wise logical not), typically cause upper bits in the data path to flip even for low-range source values. This leads to "hot stripes" in the heat maps, distinguishing data width–dependent instructions from those that are not.

These results demonstrate that data is a significant contributor to the power dissipation in this processor, although its impact depends on the instructions used by the application. As such, data width should be given some consideration in the multi threaded software energy model. For example, if necessary, a range-limited application could require that a scaling factor be applied to the model in order to account for reduced data width, in the order of 1 to 2mW per bit data width, with some exceptions for instructions such as the outliers exposed in Figure 6.

5.5. Maximizing Power Dissipation

With low-range constrained random data, the minimum energy used by any given instruction is observed. However, high-range data does not necessarily give a maximum. A test was constructed specifically to try to establish the maximum energy used by the processor's arithmetic unit and data path.

This test interleaves `lmul` (long multiply and add) instructions, which require the largest number of operands—four source (x, y, v and w), with two destination (d and e) implementing the operation described in Equation (5) and detailed in the XMOS ISA manual [May 2009]:

$$\begin{aligned} e =& \quad r[31:0] \\ d =& \quad r[63:32] \\ \text{where } r =& \quad x \times y + v + w. \end{aligned} \tag{5}$$

The even pair of test threads were loaded with the value `0x55555555` into all source registers and the odd pair with `0xaaaaaaaa`. This ensures that every bit on the input to the multiplier is flipped on every clock cycle, along with two thirds of bits in the output. This is demonstrated in Table IV, in which the Hamming weight (the number of differing bits across a pair of values) between inputs and outputs for the threads is shown.

In testing, this yielded a power dissipation of 222mW, an approximately 15% increase in the power of the instruction compared to regular 32-bit tests. Table V shows the disparity between the worst (or pathological) power dissipation of an instruction and

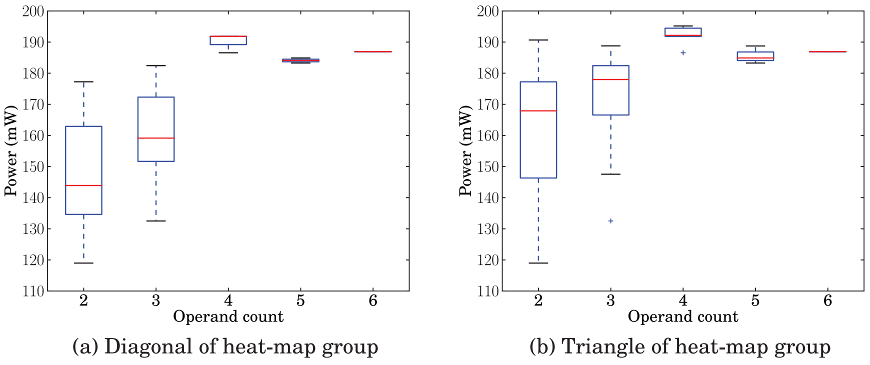


Fig. 7. Box-whisker distribution of measurements for instructions grouped by operand count.

the typical power for random data of various widths. The power used varies by up to 1.7x.

While this is an important observation, such conditions are unlikely to occur frequently. The inclusion of pathological behaviors into the model would increase the model complexity with very low likelihood of improving accuracy. As such, this behavior is not incorporated into the model. Nevertheless, awareness of pathological energy consumption is useful for debugging, and could be considered alongside other cases such as the possibility of a naïvely implemented algorithm stalling the pipeline and degrading performance with poorly scheduled sequential memory operations.

5.6. Grouping Instructions

Forming a hypothesis based on the data collected up to this point, it may be possible to group instructions by operand count (or a proxy of it, such as instruction encoding) rather than modeling each instruction individually. This subsection seeks to determine what effect this might have on model accuracy.

Figure 7 shows box plots for the five groupings of instructions, based on the operand counts that were identified from the heat maps earlier in this section. The data were generated from the same set as the heat map in Figure 5(a). Figure 7(a) is an analysis of the data along the diagonal for each group, while Figure 7(b) also includes the triangle on the diagonal of each group.

It is shown that grouping 4, 5, and 6 operand instructions should not significantly impact model accuracy. However, these groupings contain far fewer instructions than the 2 and 3 operand sets. Indeed, the remaining sets show significant variance. The data along the diagonal exhibit similar variance to when the triangle is included.

These data demonstrate that operand count can give an indication of the energy consumed by an instruction, but using it as the sole indicator of an instruction's energy consumption may lead to higher model error, depending on the types of instructions executed by the application under analysis.

5.7. Decisions Guided by Measurement Data

From the data collected and the analysis performed, decisions can be made with respect to the approach to be used to construct a model to estimate multithreaded software energy consumption. The trade-off between model accuracy and complexity must be considered so that performance can be maximized while delivering an error margin similar to previous approaches as discussed in Section 3.1. The following decisions were made:

—*Generalize the interinstruction overhead.* In Figure 5, it was shown that the power per instruction changes in the order of tens of mW depending on the instruction type, whereas the interinstruction overhead varies in the order of less than 10mW for the majority of cases. An average overhead can be used in place of individual overheads, with a low impact on accuracy, giving more flexibility to the ways in which a software energy model can be implemented and where its data can be sourced from.

—*Use instruction statistics rather than trace data.* In order to establish a reasonable trade-off between performance and accuracy, a fast, higher-level model can use instruction statistics rather than complete trace data. The performance difference between statistics and trace collection is covered in Section 3.3, where it was established that statistics collection is significantly quicker. The model can be refined and lowered to trace level if a higher level of accuracy is deemed necessary.

Further to these decisions , some practical issues must also be addressed, in particular the challenge of modeling instructions that were not directly tested by `XMProfile`. Two solutions are proposed:

—*Group instructions rather than considering their individual power.* Grouping instructions simplifies the modeling process while capturing what the data suggest to be the most significant contribution to energy consumption: the amount of data-path activity caused by the operand count. In order to establish the impact of grouping, the model will be implemented both in grouped form and at individual instruction level to allow a comparison of the error margin, as discussed in Section 5.6. Instructions not directly profiled can be accounted for by assigning them to a group based on their operand count, with the intent of giving a better power estimate than a single default for all unprofiled instructions.

—*Utilize a default energy cost for instructions that have not been profiled.* For the case in which instructions are modeled individually rather than using groups, a default value will be used. The default will be based on the 3-register instructions' average power, as it is a frequently occurring encoding. This creates a good opportunity to evaluate a preliminary model against the group model and give insight into whether the profiling needs to expanded to the complete ISA.

These sets of decisions are based on data gathered and hypotheses formed through the earlier parts of this section. They aim to build a model that is sufficiently accurate for use as a software-level energy model, particularly in relative terms, but ideally with single-digit percentage error in absolute terms. Additionally, these decisions are chosen to allow the fastest forms of instruction-set simulation to be utilized and provide transferability to other levels of abstraction with low effort.

6. MULTITHREADED SOFTWARE ENERGY MODEL: CONSTRUCTION AND EVALUATION

This section presents a model that is constructed based on the decisions detailed in the previous section. The model is then evaluated in a series of tests, with discussion of the model's performance both in terms of speed and error margin.

6.1. Overview of Model Structure

The model constructs an estimate of software energy consumption by amassing information based on the execution statistics from instruction-set simulation. These execution statistics, which can be obtained via a run of the fast `axe` or slower `xsim` simulator, provide a breakdown of all the instructions executed on an XS1 core and how many times an instruction is executed per hardware thread.

From the execution statistics, the following data are available:

—Total execution time (in cycles)
—The number of times each instruction is executed within each thread
—Number of operands used for each instruction (based on encoding)

That data can be combined with the data extracted from `XMProfile` in the previous section:

—The *base power* dissipated by the processor
—The *thread cost* dependent on how many threads are active
—*Instruction power*, as measured by the profiling runs
—An average of the *interinstruction overhead*, to represent switching between instructions, calculated from the pairwise interinstruction overheads measured during profiling

Two main variations on the model were produced: one that groups instruction power based on the operand count of instructions and a version that considers each instruction's power individually. For individual instruction modeling, the 36 directly measured ALU-based instructions have power specified, with a default power value of 150mW for any other executed instructions. With the grouped model, instructions outside this set of 36 ALU operations are assigned power values according to their operand count.

In both cases, the model uses the execution statistics to determine how much time the processor spends executing each instruction as well as how many threads are active at that time. The thread activity is not a precise calculation because a full trace is not captured. Instead, activity is estimated based on the distribution of instructions executed by each thread over the complete runtime.

Equation (6) describes the energy of a program E_p using a similar method to Equation (3). However, time is considered explicitly in this new model. This gives us an energy value rather than power. In addition, this makes it possible to account for idle time, wherein no instructions are executed because no threads are active. To achieve this, the number of cycles with no active threads N_{idle} is measured, then multiplied by the clock period T_{clk} and base power P_{base}, which is the power dissipated when the processor is idle.

Next, to account for pipeline activity, for each possible number of concurrent threads up to the maximum, N_t, a multiplier M_t is applied for that level of concurrency. Finally, for all instructions in the Instruction Set Architecture (ISA), each instruction i at concurrency level t is assigned an instruction power P_i that is scaled by a constant overhead O to account for interinstruction overheads. The base power is then added and the result is multiplied by the clock period and the number of times this instruction is executed at concurrency level t, $N_{i,t}$. This gives an estimate for the total energy consumed over the runtime of the application within the processor core, accounting for potential variation in concurrency level over that time, as well as idle time:

$$E_{\text{p}} = P_{\text{base}} N_{\text{idle}} T_{\text{clk}} + \sum_{t=1}^{N_t} \sum_{i \in \text{ISA}} \big((M_t P_i O + P_{\text{base}}) N_{i,t} T_{\text{clk}} \big). \tag{6}$$

For grouped instructions, the instruction power P_i is replaced with a lookup against which group the instruction belongs to, $P_{G(i)}$.

These models can be used on instruction execution statistics, alongside the data collected for the XS1-L processor, to produce a value representing the estimated energy consumption in joules of the simulated multithreaded program p.

The XS1-L differs from various modeled processors in that it has no caches. Cache hierarchies therefore do not need to be considered, which simplifies one aspect of modeling. However, there are other complexities in the processor that require new approaches in order to account for them. In particular, idle time is captured explicitly, which is appropriate for an I/O-centric, event-driven processor. Further, the introduction of threading level into the model is necessary for sufficiently accurate energy estimation.

Without the M_t term, instruction power would be mispredicted by an additional 10% or more for single-threaded sections of a program, where $M_1 = 0.25$ as used in our modeling. A smaller error would also exist for 2- or 3-threaded execution. An alternative method of accounting for this would be to have separate costs for each instruction at each level of concurrency, bringing the model closer to prior single-threaded work such as Tiwari et al. [1994b]. The M_t parameterized approach is preferred because it reduces the profiling effort required. In addition, the parameterized model is a closer representation of the processor's pipeline characteristics; higher-level analysis can also benefit from the model exposing threading in this way.

Profiling and modeling techniques such as those outlined by Contreras and Martonosi [2005] rely on hardware performance counters. This approach is not suitable for embedded processors, which do not have such counters, or when accessing such counters could disrupt the intended operation of a tightly timed real-time program. Therefore, it is important to provide an alternative method that overcomes these restrictions but provides a similar level of detail in simulation. The approach presented here seeks to fill this gap.

6.2. Testing the Model

To test the model, a suite of benchmarks was selected and run through the axe XS1 Instruction Set Simulation; the statistics from it were passed through the model to estimate the benchmark's energy consumption.

The benchmarks represent generic workloads as well as workloads typical of the XS1-L processor. The list of benchmarks is described in Table VI. These benchmarks were selected to utilize various processor features, including shared memory, message passing, various degrees of parallelism, integer arithmetic, string processing, and fused operations such as multiply-accumulate. The proportion of instructions (based on the number of times each instruction is executed) that are directly modeled is noted in the rightmost column of the table, to give an indication of how complete the model is in relation to the particular test's distribution of instructions. A memory-intensive benchmark will rely more on the default energy model value due to the current model data not directly handling memory instructions.

Tests were run for 0.4s to achieve a simulation time of less than 1min while providing sufficient runtime for thousands of power samples to be collected. This simulation time is a reasonable length of time for a programmer to wait for an energy figure compared to the effort of instrumenting and measuring physical hardware. If the programmer does not have hardware access at the time, longer simulation may be acceptable. Ultimately, runtime helps to establish what this modeling method can achieve when considering the needs of software developers. Longer and shorter test runs were also performed on several benchmarks to verify that the model and energy readings did not diverge over extended execution.

6.3. Evaluation

This evaluation presents a comparison between energy estimations from the model and measurements taken on real hardware when running the benchmarks previously discussed in Section 6.2. Both individual instruction (Equation (6)) and grouped instruction models are examined.

Table VI. List of Benchmarks Used to Evaluate Energy Model Accuracy

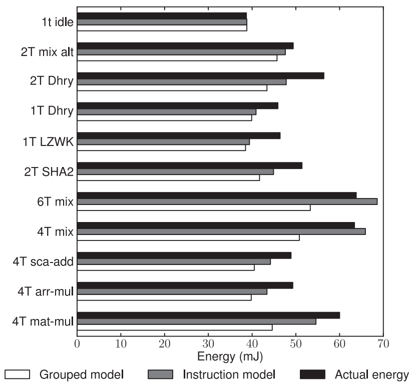
Name	Description	Utilized libraries or software	Number of threads	Proportion of instructions profiled directly
Idle	A single thread that sleeps	*None*	1	10%
Dhry	Dhrystone benchmark, run once, or twice concurrently	Dhrystone [Weicker 1984]	1, 2	33%
LZWK	Modified LZW for low-memory and real-time, single thread	Own, modified LZW [Welch 1984]	1	42%
SHA2	SHA2 hash function with “client” and “server” threads	`sc_crypto`	2	68%
Sca-add	Matrix addition with a scalar- value, shared memory	`sc_matrix`	4	39%
Arr-mul	Piecewise multiplication of two arrays	`sc_matrix`	4	36%
Mat-mul	Matrix cross-product	`sc_matrix`	4	36%
Mix	Simple audio mixing, 24-bit samples, 4- and 6-input channels	`sc_matrix`	4, 6	60%
Mix alt	Two-channel audio mixing, more advanced approach	`sc_audio_mixer`	2	32%
Matrix, crypto, and mixer libraries available from https://github.com/xcore/				

In Figure 8(a), both models are seen to be calibrated well against a single idle thread. Figure 8(b) shows that the worst-case error for the grouped model is –26%, while for the instruction level model it is −16%. The average error is –16% for the grouped model and −7% for the instruction-level model. The Spearman rank of the instruction model error and the proportion of profiled instructions is low, at 0.09 with a 95% confidence interval of 79%, suggesting that exhaustively profiling all instructions will not lead to a vastly improved model.

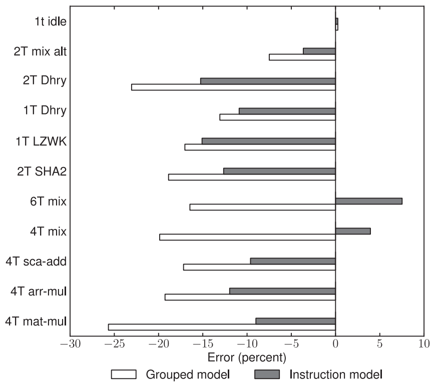
Given the consistent underestimation of the grouped model, it should be possible to improve the error margin to approximately +/−10% or better. However, this cannot be achieved with a naïve flat offset, as this would skew idle energy accuracy, which is particularly important in an event-driven system that may have a low duty cycle (long periods of inactivity between external stimuli such as network events, user input, and so on).

In Sections 5.4 and 5.6, it was observed that instructions that perform value extension, or always produce low-range or Boolean results, are the lowest-power instructions. Working under the hypothesis that these instructions contribute the most to the error in the grouped instruction approach, removing them from the 2- and 3-operand sets and putting them into a special set may avert this behavior. Figure 9 shows that this additional split reduces the variance of the groupings when compared to the initial groupings chosen and analyzed in Figure 7.

Further examination and experimentation with the values chosen for the instruction groups, particularly when compared against real-world applications, could be a route to refinement. The same approach could also be used to further improve the instruction-level model. Linear regression analysis can be used to estimate the energy of unprofiled instructions, combining the existing dataset and properties shared

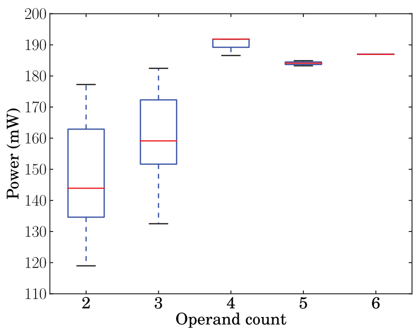


(a) Results of benchmarks for the models versus actual device measurement.

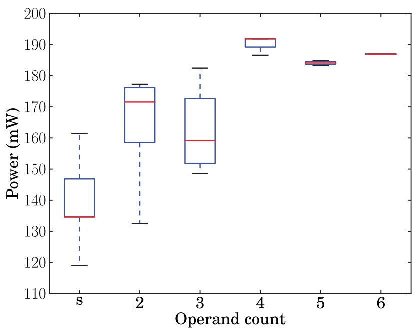


(b) Accuracy of the two model approaches compared to observed device energy.

Fig. 8. Benchmark energy results and error margins.



(a) Original grouping



(b) Additional group, s

Fig. 9. Box-whisker comparison between original groupings and with lower-power instructions separated into a special group s.

between instructions, such as instruction width and operand count. This approach has been used successfully on other architectures [Lee et al. 2001; Nunez-Yanez and Lore 2013].

The data from our testing clearly indicate that the simplification achieved by the grouped model does not outweigh the additional error that it introduces. Our per-instruction model is therefore the better method for this processor.

7. FUTURE WORK

The model presented in this article demonstrates that, with a revised profiling approach and analysis of the instruction set simulation data, a hardware multithreaded pipeline can be modeled in a similar way to previous ISA-level energy models. The empirical analysis of ALU instructions is the starting point of this model. Additional instructions can be profiled in the same way, with some modification to the `XMProfile` test harness, but not all instructions in the ISA can be executed independently in a way that would be typical of their general usage in order to establish a power figure. This presents an opportunity to use linear regression to express the energy characteristics of a wider range of the ISA, as identified in the review of previous works in Section 3.1. A parameterized version of the model would be needed, upon which linear regression analysis could be performed and will require more complex instruction sequences to be profiled within the existing measurement framework. At this level of complexity, benchmarks and real-world applications become feasible as candidates for profiling and statistical analysis.

Looking beyond the processor core, a system-level view may be more desirable. Depending on the system, the processor core may account for only a fraction of total energy [Fonseca et al. 2008], thus motivating expansion of the model in this direction. This requires analysis of power dissipated in the 3.3V voltage domain of the processor, in addition to the 1.0V measurements already taken. In the XS1 architecture, the use of special I/O and communication instructions, rather than memory-mapped peripheral interfaces, should make this analysis more intuitive and comparatively simple to perform, although nondeterministic behavior from external stimuli is also introduced to the system at this level, which may affect the combinations of active threads.

Given that the model does not rely on trace data but instead on execution statistics, it could conceivably be used with statistics obtained at a higher level than ISA simulation, for example, static analysis of object binaries or even at the source level. First steps have been taken in this direction with static analysis of assembly listings [Liqat et al. 2013]. This could lead to integration of the multithreaded energy model into the compiler toolchain, thus providing compile-time energy consumption information.

8. CONCLUSIONS

The energy characteristics of a hardware multithreaded microprocessor architecture differ from a more traditional microprocessor; the effects of this on software energy modeling have been identified, discussed, and subsequently accounted for in a new instruction set–level software energy model. Specifically, we have expanded on prior work that models software energy on single-threaded architectures, taking into account the more complex behaviors present in a hardware multithreaded and event-driven architecture, such as idle time, active thread count, and the interactions between varying numbers of active threads.

An analysis of the behavior of the multithreaded pipeline in the XS1-L processor was presented along with a hardware/software framework, `XMProfile`, for acquiring instruction-level power readings suitable for use as the base of a software energy model. A method was presented that allows precise control of the processor pipeline to guarantee instruction sequencing in the XS1-L multithreaded pipeline.

It was shown that the energy cost of individual instructions differs by up to 1.7x. Interinstruction overheads are more complex to predict in this type of architecture, thus are a barrier to some instruction-level energy modeling techniques. However, it is not necessary to consider them on a per-clock basis on account of their low significance when compared to factors such as the specific instructions that are executing and the number of active threads.

Using data extracted with `XMProfile`, it was then shown how execution statistics from simulation runs can be used to model the energy of a set of benchmarks with a typical error of 7% compared to actual hardware energy measurements. An alternative grouped instruction model was proposed and demonstrated, but the performance benefits were shown to be low compared to a loss of accuracy of several percentage points. Execution statistics were used rather than instruction traces, reducing simulation time by two orders of magnitude.

Future work was proposed, such as how the model can be improved further by refining various terms in the model based on feedback from benchmarks, and by accounting for a wider range of instructions as well as activity in the 3.3V domain. The statistics-based multithreaded software energy model presented here can be used with data from higher levels of abstraction than instruction-set simulation. Finally, the methods for building an ISA-level energy model of the XS1-L, as demonstrated in this article, provide a template that allows energy models for other multithreaded architectures to be built.

ACKNOWLEDGMENTS

We thank Henk Muller and the team at XMOS for the supply of hardware and technical support. A special thank you to James Pallister for providing *matplotlib* expertise.

REFERENCES

Todd Austin. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer* 35, 2, 59–67.

Daniel Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel*. Oreilly & Associates Inc., Sebastopol, CA.

David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News* 28, 2, 83–94.

David R. Butenhof. 1997. *Programming with POSIX threads*. Addison-Wesley Professional, New York, NY.

Gilberto Contreras and Margaret Martonosi. 2005. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED'05)*. IEEE, 221–226.

Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. 2008. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Symposium of Operating Systems Design and Implementation (OSDI'08)*. 323–328.

Mostafa E. a. Ibrahim, Markus Rupp, and Hossam a. H. Fahmy. 2008. Power estimation methodology for VLIW digital signal processors. In *Proceedings of the 2008 42nd Asilomar Conference on Signals, Systems and Computers*. IEEE, 1840–1844.

Intel Corporation. 2003. Intel Hyper-Threading Technology Technical User's Guide.

Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro* 25, 2, 21–29.

Sheayun Lee, Andreas Ermedahl, and Sang Lyul Min. 2001. An accurate instruction-level energy consumption model for embedded RISC processors. *ACM SIGPLAN Notices* 36, 8, 1–10.

Umer Liqat, Steven Kerrison, Serrano Alejandro, Kyriakos Giorgiou, Pedro Lopez-Garcia, Neville Grech, Manuel V. Hermenegildo, and Kerstin Eder. 2013. Energy consumption analysis of programs based on XMOS ISA-level models. In *23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*. 72–90.

David May. 2009. *XMOS XS1 Instruction Set Architecture*.

David May, Ali Dixon, Ayewin Oung, Henk Muller, and Mark Lippett. 2008. XS1-L System Specification.

Microsoft Corporation. 2012. About Processes and Threads (Windows). (2012). Retrieved March 22, 2015 from http://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx.

Jose Nunez-Yanez and Geza Lore. 2013. Enabling accurate modeling of power and energy consumption in an ARM-based system-on-chip. *Microprocessors and Microsystems* 37, 3, 319–332.

Gang Qu, Naoyuki Kawabe, Kimiyoshi Usami, and Miodrag Potkonjak. 2000. Function-level power estimation methodology for microprocessors. In *Proceedings of the 37th Conference on Design Automation (DAC'00),* 810–813.

Kaushik Roy and Mark C. Johnson. 1997. Software design for low power. In *Low Power Design in Deep Submicron Electronics*. Kluwer Academic Publishers, Dordrecht, The Netherlands, Chapter 6, 433–460.

Jeffry T. Russell and Margarida F. Jacome. 1998. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*. IEEE Computer Society, 328–333.

Mariagiovanna Sami, Donatella. Sciuto, Cristina Silvano, and Vittorio Zaccaria. 2002. An instruction-level energy model for embedded VLIW architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21, 9, 998–1010.

Yakun Sophia Shao and David Brooks. 2013. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'13)*. IEEE, 389–394.

Sim-Panalyzer. 2004. *Sim-Panalyzer 2.0 Reference Manual*.

Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. 2001. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proceedings of the International Workshop on Power and Timing Modeling Optimization and Simulation (PATMOS'01)*.

Vivek Tiwari, Sharad Malik, and Andrew Wolfe. 1994a. Compilation techniques for low energy: An overview. In *Proceedings of the IEEE Symposium on Low Power Electronics, 1994. Digest of Technical Papers*. IEEE, 38–39.

Vivek Tiwari, Sharad Malik, and Andrew Wolfe. 1994b. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* 2, 4, 437–445.

Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. 1996. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing Systems for Signal,Image, and Video Technology* 13, 2–3, 223–238.

Reinhold P. Weicker. 1984. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM* 27, 10, 1013–1030.

Terry A. Welch. 1984. A technique for high-performance data compression. *Computer* 17, 6, 8–19.

Received September 2013; revised June 2014; accepted November 2014