Determining Application-Specific Peak Power and Energy Requirements for Ultra-Low-Power Processors

HARI CHERUPALLI, University of Minnesota
HENRY DUWE, WEIDONG YE, and RAKESH KUMAR, University of Illinois
JOHN SARTORI, University of Minnesota

Many emerging applications such as the Internet of Things, wearables, implantables, and sensor networks are constrained by power and energy. These applications rely on ultra-low-power processors that have rapidly become the most abundant type of processor manufactured today. In the ultra-low-power embedded systems used by these applications, peak power and energy requirements are the primary factors that determine critical system characteristics, such as size, weight, cost, and lifetime. While the power and energy requirements of these systems tend to be application specific, conventional techniques for rating peak power and energy cannot accurately bound the power and energy requirements of an application running on a processor, leading to overprovisioning that increases system size and weight. In this article, we present an automated technique that performs hardware–software coanalysis of the application and ultra-low-power processor in an embedded system to determine application-specific peak power and energy requirements. Our technique provides more accurate, tighter bounds than conventional techniques for determining peak power and energy requirements. Also, unlike conventional approaches, our technique reports guaranteed bounds on peak power and energy independent of an application's input set. Tighter bounds on peak power and energy can be exploited to reduce system size, weight, and cost.

CCS Concepts: • **Hardware → Power estimation and optimization; • Computing methodologies → Modeling and simulation; • Hardware; • Computer systems organization →** *Embedded and cyber-physical systems;* **• Software and its engineering;**

Additional Key Words and Phrases: Peak power analysis, application-specific hardware, ultra-low-power processors, hardware–software coanalysis, Internet of Things

The main idea of this work on determining peak power and energy requirements first appeared in the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17). This manuscript extends that work with a detailed description and analysis of the underlying hardware–software coanalysis. In particular, it provides motivation and examples of cases where a naïve coanalysis approach is intractable and how the proposed scalable approach handles these cases. Several of the authors' other prior works use the hardware-software coanalysis technique. However, these works either use and describe a nonscalable coanalysis ("Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems" in ISCA'16) or do not detail or analyze the underlying hardware–software coanalysis in terms of scalability when applied on complex applications as this manuscript does ("Enabling Effective Module-oblivious Power Gating for Embedded Processors" in HPCA'17, "Bespoke Processors for Applications with Ultra-low Area and Power Constraints" in ISCA'17, and "Software-based Gate-level Information Flow Security for IoT Systems" to appear in MICRO'17). This article also provides analysis and results of cost savings enabled by tighter bounds on peak power and energy for several real batteries and systems.

Authors' addresses: H. Cherupalli, 200 Union St. S.E., Keller Hall 4-174, Minneapolis, MN 55455; email: cheru007@umn.edu; H. Duwe, 613 Morrill Rd, 321 Durham Center, Ames, IA 50011; email: duwe@iastate.edu; W. Ye, 1308 West Main Street, 210 Coordinated Science Laboratory, Urbana, IL 61801; email: wye5@illinois.edu; R. Kumar, 1308 West Main Street, 208 Coordinated Science Laboratory, Urbana, IL 61801; email: rakeshk@illinois.edu; J. Sartori, 200 Union St. S.E., Keller Hall 4-163, Minneapolis, MN 55455; email: jsartori@umn.edu.

https://doi.org/10.1145/3148052

ACM Reference format:
Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Determining Application-Specific Peak Power and Energy Requirements for Ultra-Low-Power Processors. *ACM Trans. Comput. Syst.* 35, 3, Article 9 (December 2017), 33 pages.
https://doi.org/10.1145/3148052

1 INTRODUCTION

Ultra-low-power (ULP) processors have rapidly become the most abundant type of processor in production today. New and emerging power- and energy-constrained applications—such as the Internet of Things (IoT), wearables, implantables, and sensor networks—have already caused production of ULP processors to exceed that of personal computers and mobile processors (Greenough 2015). The 2015 ITRS report projects that these applications will continue to rely on simple single-core, ultra-low-power processors in the future, will be powered by batteries and energy harvesting, and will have even tighter peak power and energy constraints than the power- and energy-constrained ULP systems of today (ITRS 2015). Unsurprisingly, low-power microcontrollers and microprocessors are projected to continue being the most widely used type of processor in the future (IC Insights 2015; Greenough 2015; Evans 2011; Press 2014).

ULP systems can be classified into three types based on the way they are powered (Calhoun et al. 2010). As illustrated in Figure 1, some ULP systems are powered directly by energy harvesting (Type 1), while some are battery powered (Type 3). Another variant is powered by a battery and uses energy harvesting to charge the battery (Type 2).

For each of the above classes, the size of energy harvesting and/or storage components determine the form factor, size, and weight. Consider, for example, the wireless sensor node shown in Figure 2 (National Instruments 2013). The two largest system components that predominantly determine the overall system size and weight are the energy harvester (solar cell) and the battery.

Going one step further, since the energy harvesting and storage requirements of a ULP system are determined by its power and energy requirements, the peak power and energy requirements of a ULP system are the primary factors that determine critical system characteristics such as size, weight, cost, and lifetime (Calhoun et al. 2010). In Type 1 systems, peak power is the primary constraint that determines system size, since the power delivered by harvesters is proportional to their size. In these systems, harvesters must be sized to provide enough power, even under peak load conditions. In Type 3 systems, peak power largely determines battery life, since it determines the *effective battery capacity* (Buchmann 2016). As the rate of discharge increases, effective battery capacity drops (Buchmann 2016; Furset and Hoffman 2011). This effect is particularly pronounced in ULP systems, in which near-peak power is consumed for a short period of time, followed by a much longer period of low-power sleep, since pulsed loads with high peak current reduce effective capacity even more drastically than sustained current draw (Furset and Hoffman 2011).

In Types 2 and 3 systems, the peak energy requirement matters as well. For example, energy harvesters in Type 2 systems must be able to harvest more energy than the system consumes, on average. Similarly, battery life and effective capacity are dependent on energy consumption (i.e., average power) (Furset and Hoffman 2011). Figure 3 summarizes how peak power and energy requirements impact sizing parameters for the different classes of ULP systems.

Finally, Tables 1 and 2 list the energy and power densities for different types of batteries and energy harvesters, respectively. These data provide a rough sense of how size and weight of a ULP system scale are based on peak energy and power requirements. A tighter bound on the peak

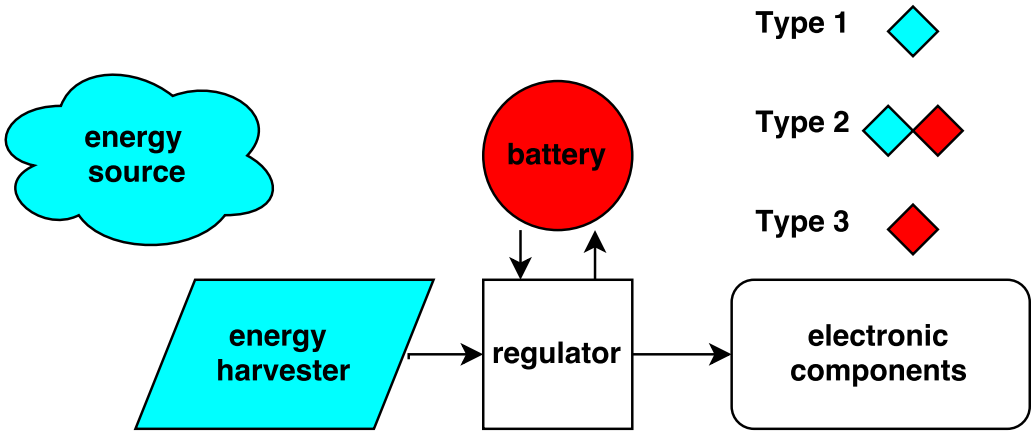


Fig. 1. ULP systems are commonly powered by energy harvesting, battery, or a combination of the two, in which harvesters are used to charge the battery.

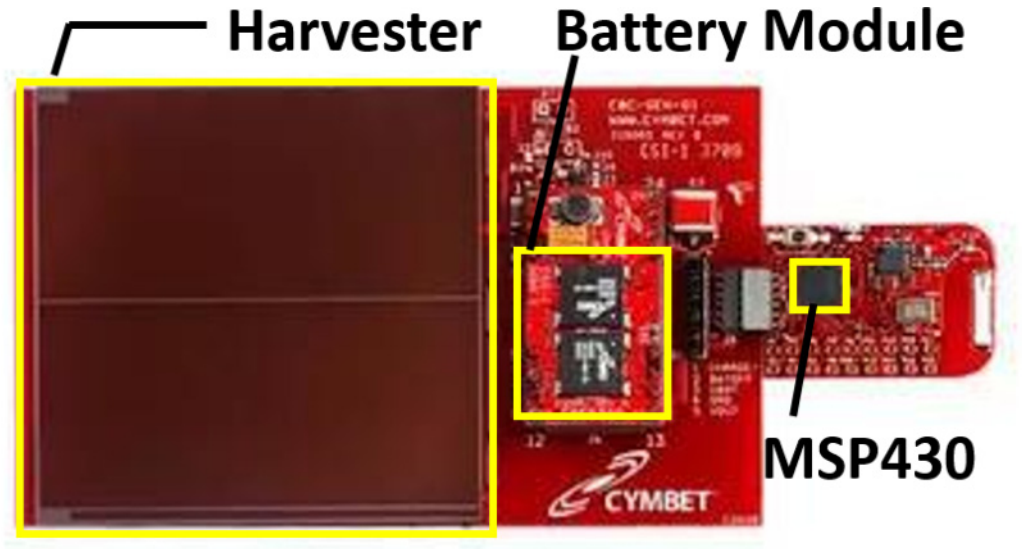


Fig. 2. In most ULP systems, like this wireless sensor node, the size of the battery and/or energy harvester dominates the total system size.

Table 1. Specific Energy and Energy Density for Different Battery Types (bat 2015)

Battery Type	Specific Energy [J/g]	Energy Density [MJ/L]
Li-ion	460	1.152
Alkaline	400	0.331
Carbon-zinc	130	1.080
Ni-MH	340	0.504
Ni-cad	140	0.828
Lead-acid	146	0.360

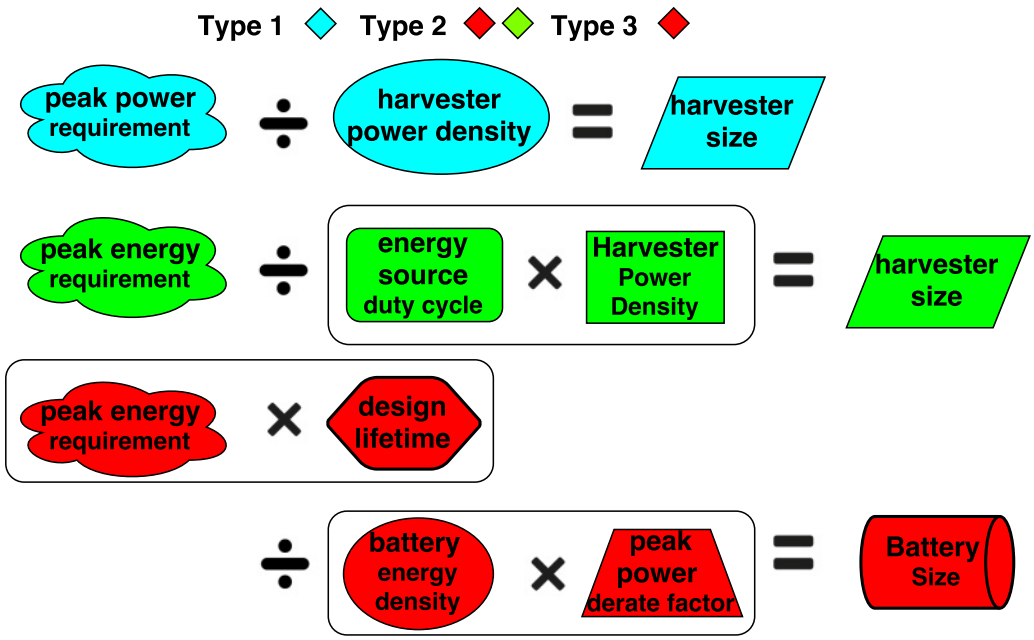


Fig. 3. Harvester and battery size calculations for Types 1, 2, and 3 ULP systems depend on peak power and energy requirements.

Table 2. Power Density for Different Types of Energy Harvesters (Paradiso and Starner 2005)

Harvester type	Power Density
Photovoltaic (sun)	100 mW/cm^2
Photovoltaic (indoor)	100 $\mu W/cm^2$
Thermoelectric	60 $\mu W/cm^2$
Ambient airflow	1 mW/cm^2

power and energy requirements of a ULP system can result in a roughly proportional reduction in size and weight.

How are Peak Power and Energy Determined Today?

There are several possible approaches to determining the peak power and energy requirements of a ULP processor (Figure 4).[1] The most conservative approach involves using the processor design specifications provided in data sheets. These specifications characterize the peak power that can be consumed by the hardware at a given operating point and can be directly translated into a bound on peak power. This bound is conservative because it is not application specific; however, it is safe for any application that might be executed on the hardware. A more aggressive technique for determining peak power or energy requirements is to use a peak power or energy stressmark. A stressmark is an application that attempts to activate the hardware in a way that maximizes peak power or energy. A stressmark may be less conservative than a design specification, since it may

[1]Peak power and energy are sometimes referred to as worst-case power and energy.

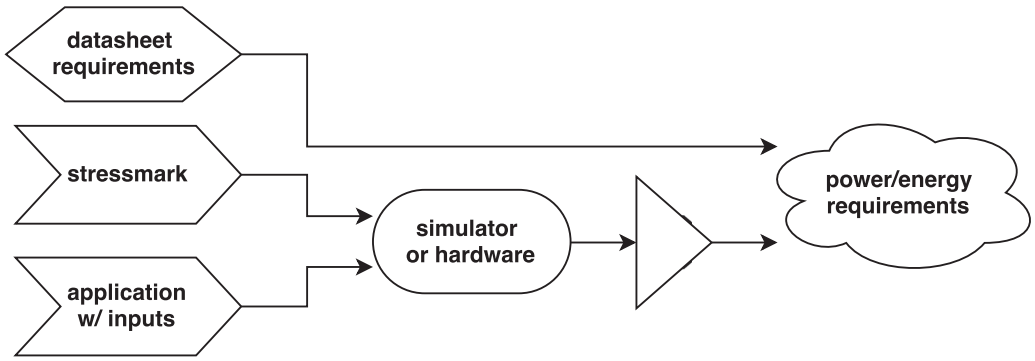
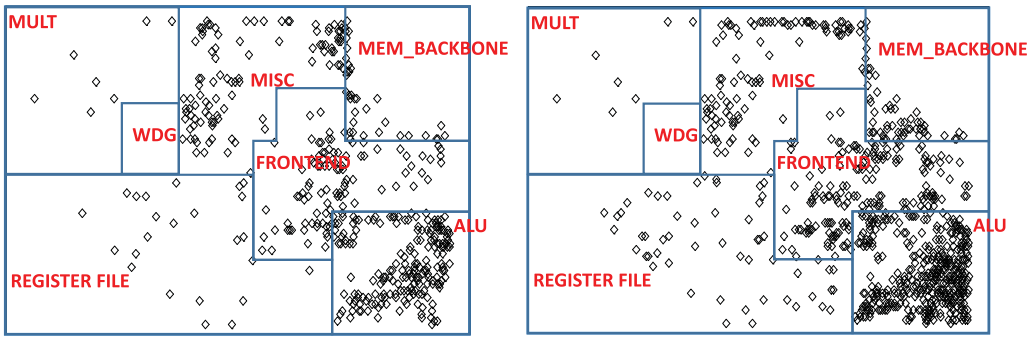


Fig. 4. The conventional methodology for sizing energy harvesting and storage components involves determining peak power and energy requirements for a processor and selecting components that will provide enough power and energy to satisfy the requirements over the lifetime of the system.



(a) Active gates at the peak cycle for tHold (452 gates). **(b)** Active gates at the peak cycle for PI (743 gates).

Fig. 5. Different applications can have different activity profiles, resulting in peak power and energy requirements that are application specific.

not be possible for an application to exercise all parts of the hardware at once. The most aggressive conventional technique for determining peak power or energy of a ULP processor is to perform application profiling on the processor by measuring power consumption while running the target application on the hardware. However, since profiling is performed with specific input sets under specific operating conditions, peak power or energy bounds determined by profiling might be exceeded during operation if application inputs or system operating conditions are different than during profiling. To ensure that the processor operates within its peak power and energy bounds, a guardband is applied to profiling-based results.

Our Proposal: Determining Application-Specific Peak Power and Energy Requirements
Most ULP embedded systems run the same application or computation over and over in a compute/sleep cycle for the entire lifetime of the system (EEMBC 2017). As such, the power and energy requirements of embedded ULP processors tend to be application specific. This is not surprising, considering that different applications exercise different hardware components at different times, generating different application-specific loads and power profiles. For example, Figure 5(a) and Figure 5(b) show the active (toggling) gates for two different applications (tHold and PI; see Table 3) during the cycles in which peak power is expended for each application. These figures were generated by running gate-level simulations of the applications on openMSP430 (Girard

2013) and marking all gates that toggled in the cycle in which each benchmark expended its peak power. The figures show that PI exercises a larger fraction of the processor than tHold at its peak, leading to higher peak power. However, while the peak power and energy requirements of ULP processors tend to be application specific, many conventional techniques for determining peak power and energy requirements for a processor are not application specific (e.g., design-based and stressmark-based techniques). Even in the case of a profiling-based technique, guardbands must be used to inflate the peak power requirements observed during profiling, since it is not possible to generate bounds that are guaranteed for all possible input sets. These limitations prevent existing techniques from accurately bounding the power and energy requirements of an application running on a processor, leading to overprovisioning that increases system size and weight.

In this article, we present a novel technique that determines application-specific peak power and energy requirements based on hardware–software coanalysis of the application and ultra-low-power processor in an embedded system. Our technique performs a novel, scalable symbolic simulation of an application on the processor netlist in which unknown logic values (Xs) are propagated for application inputs.[2] This allows us to identify gates that are guaranteed to not be exercised by the application for any input. This, in turn, allows us to bound the peak power and energy requirements for the application. The peak power and energy requirements generated by our technique are guaranteed to be safe for all possible inputs and operating conditions. Our technique is fully automated and provides more accurate, tighter bounds than conventional techniques for determining peak power and energy requirements. Our article makes the following contributions.

- We present an automated technique based on a novel, scalable symbolic simulation approach that takes an embedded system's application software and processor netlist as inputs and determines application-specific peak power and energy requirements for the processor that are guaranteed to be valid for all possible application inputs and operating conditions. This is the first approach to use symbolic simulation to determine peak power and energy requirements for an application running on a processor.
- We show that the application-specific peak power and energy requirements determined by our technique are more accurate, and therefore less conservative, than those determined by conventional techniques. On average, the peak power requirements generated by our technique are 27%, 26%, and 15% lower than those generated based on design specifications, a stressmark, and profiling, respectively, and the peak energy requirements generated by our technique are 47%, 26%, and 17% lower, respectively. Reduction in the peak power and energy requirements of a ULP processor can be leveraged to improve critical system metrics such as size and weight.
- Our technique can be used to guide optimizations that target and reduce the peak power of a processor. Optimizations suggested by our technique reduce peak power by up to 10% for a set of embedded applications.

2 A CASE FOR APPLICATION-SPECIFIC, INPUT-INDEPENDENT PEAK POWER AND ENERGY REQUIREMENTS

We measured peak power consumption for a sample set of ULP benchmark applications (see Table 3) running on an MSP430 F1610 processor.[3] Benchmark applications were run repeatedly

[2]Peak power and energy analyses can be offered as a cloud compilation service by the hardware system vendor in settings in which the application developer does not have access to the processor description (ARM Mbed 2017; Cloud Compiling 2013; National Instruments 2016).

[3]MSP430 is one of the most popular processors used in ULP systems (Borgeson 2012; Wikipedia 2016).

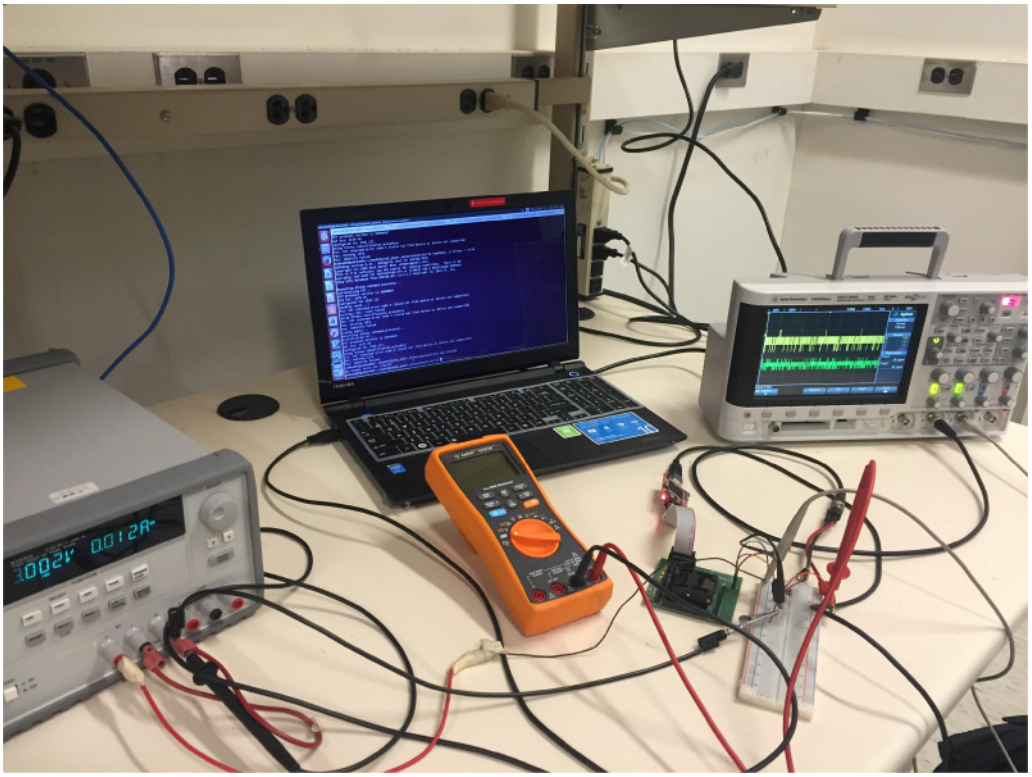
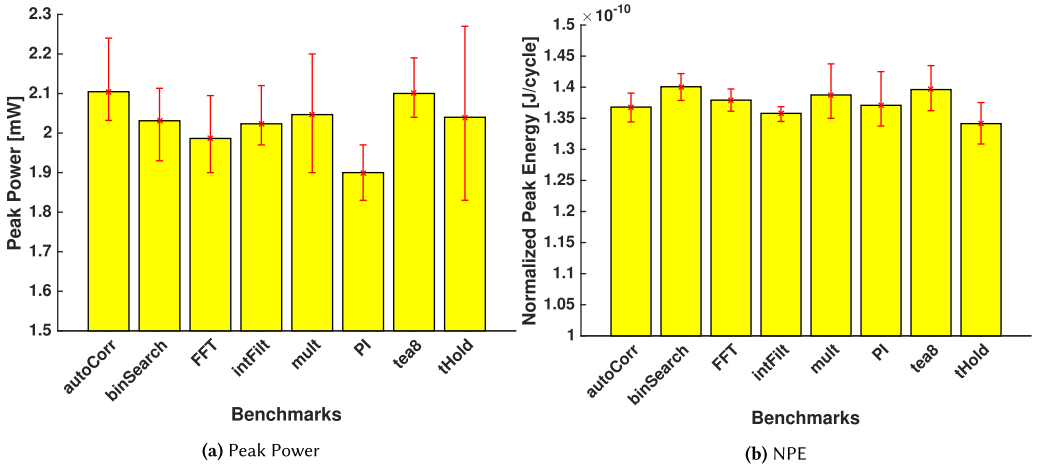


Fig. 6. The test setup used to measure peak and average power on a ULP processor (MSP430).

with different inputs at an operating frequency of 8 MHz while sampling the voltage and current of the processor at a rate of 10 MHz using an InfiniiVision DSO-X 2024A oscilloscope, to ensure at least one sample per cycle. Power is calculated as the product of voltage and current. Figure 6 shows our test setup.

Figure 7(a) compares the peak power observed for different applications. The results show that peak power can be different for different applications. Thus, peak power bounds that are not application specific will overestimate the peak power requirements of applications, leading to overprovisioning of energy harvesting and storage components that determine system size and weight. Figure 7(a) also shows that the peak power requirements of applications are significantly lower than the rated peak power of the chip (4.8 mW); thus, using design specifications to determine peak power requirements can lead to significant overprovisioning and inefficiency. The figure also confirms that peak power of an application depends on application inputs and can vary significantly for different inputs. This means that profiling cannot be relied on to accurately determine the peak power requirement for a processor, since not all input combinations can be profiled and the peak power for an unprofiled input could be significantly higher than the peak power observed during profiling. Since input-induced variations change peak power by over 25% for these applications (Figure 7(a)), a profiling-based approach for determining peak power requirements should apply a guardband of at least 25% to the peak power observed during profiling.

For energy-constrained ULP systems, such as those powered by batteries (Types 2 and 3), peak energy and peak power determine the size of energy harvesting and storage components (Section 1). Thus, it is also important to determine an accurate bound on the peak energy



(a) Peak Power (b) NPE

Fig. 7. The peak power and normalized peak energy (normalized to an application's runtime in cycles) of a ULP processor are different for different applications and different inputs. The bars represent average across all inputs; error bars show the range of input-induced peak and average power variations. Measured variation between multiple runs of the same application and same input is less than 2%.

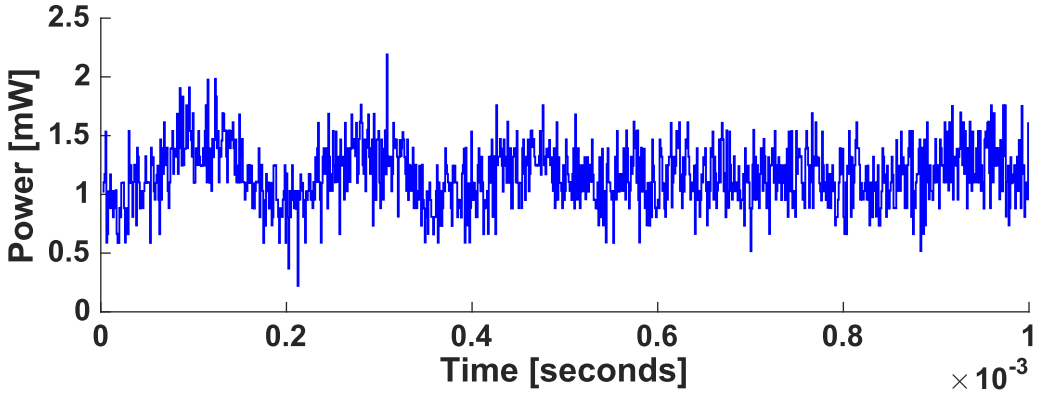


Fig. 8. Measured instantaneous power of MSP430F1610 for the *mult* benchmark is significantly lower, on average, than both the rated and observed peak power for the application.

requirements of a ULP processor. Figure 8 shows the instantaneous power profile for an application (*mult*), demonstrating that, on average, instantaneous power can be significantly lower than peak power. Therefore, we can more accurately determine the optimal sizing of components in an energy-constrained system by generating an accurate bound on peak energy rather than conservatively multiplying peak power by execution time.

Figure 7(b) characterizes the peak energy, normalized to application runtime in cycles, for different applications and input sets, showing that the maximum rate at which an application can consume energy is also application and input dependent. Therefore, conventional techniques for determining the peak energy requirements of a ULP processor have the same limitations as conventional techniques for determining peak power requirements. In both cases, the limitations of conventional techniques require overprovisioning that can substantially increase system size and weight.

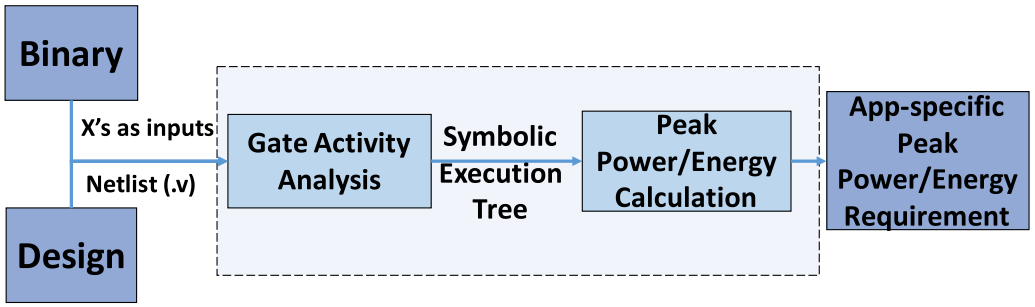


Fig. 9. Our technique performs input-independent activity analysis that enables determination of accurate peak power and energy requirements for a ULP processor.

In the next section, we describe a novel technique for determining the peak power and peak energy requirements of a ULP processor that is application specific yet also input independent.

3 APPLICATION-SPECIFIC INPUT INDEPENDENT PEAK POWER AND ENERGY

Figure 9 provides an overview of our technique for determining application-specific peak power and energy requirements that are input independent. The inputs to our technique are the application binary that runs on a ULP processor and the gate-level netlist of the ULP processor. The first phase of our technique, described in Section 3.1, is an activity analysis that uses a novel symbolic simulation technique to efficiently characterize all possible gates that can be exercised for all possible execution paths of the application and all possible inputs. This analysis also reveals which gates can *never* be exercised by the application. Based on this analysis, we perform input-independent peak power (Section 3.2) and energy (Section 3.3) calculations to determine the peak power and energy requirements for a ULP processor.

3.1 Input-Independent Gate Activity Analysis

Since the peak power and energy requirements of an application can vary based on application inputs, a technique that determines application-specific peak power requirements must bound peak power for all possible inputs. Exhaustive profiling for all possible inputs is not possible for most applications; thus, we have created a novel approach for activity analysis that uses unknown logic values (Xs) for inputs to efficiently characterize activity for all possible inputs with minimum simulation effort.

Our technique, described in Algorithm 1, is based on symbolic simulation (Bryant 1991) of an application binary running on the gate-level netlist of a processor, in which Xs are propagated for all signal values that cannot be constrained based on the application. When the simulation begins, the states of all gates and memory locations that are not explicitly loaded with the binary are initialized to Xs. During simulation, all input values are replaced with Xs by our simulator. As simulation progresses, the simulator dynamically constructs an execution tree describing all possible execution paths through the application. If an X symbol propagates to the inputs of the program counter (PC) during simulation, indicating an input-dependent control sequence, a branch is created in the execution tree. Normally, the simulator pushes the state corresponding to one execution path onto a stack for later analysis and continues down the other path. However, a path is not pushed to the stack or resimulated if it has already been simulated (i.e., if the simulator has seen the branch (PC) before and the processor state is the same as it was when the branch was previously encountered). This allows Algorithm 1 to analyze programs with input-dependent loops. When simulation down one path reaches the end of the application, an unsimulated state is loaded from the last

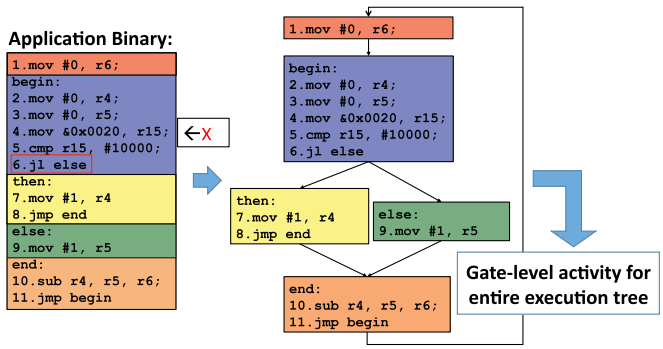


Fig. 10. Symbolic input-independent gate activity analysis involves using unknown logic values for application inputs to characterize application-induced processor behavior for all possible executions of an application.

ALGORITHM 1: Input-Independent Gate Activity Analysis

```
1.  Procedure Create Symbolic Execution Tree(app_binary, design_netlist)
2.  Initialize all memory cells and all gates in design_netlist to X
3.  Load app_binary into program memory
4.  Propagate reset signal
5.  s ← State at start of app_binary
6.  Symbolic Execution Tree T.set_root(s)
7.  Stack of unprocessed execution paths, U.push(s)
8.  while U != ∅ do
9.      e ← U.pop()
10.     while e.PC_next != X and !e.END do
11.         e.set_inputs_X() // set all peripheral port inputs to Xs
12.         e' ← propagate_gate_values(e) // simulate this cycle
13.         e.annotate_gate_activity(e,e') // annotate activity in tree
14.         e.add_next_state(e') // add to execution tree
15.         e ← e' // process next cycle
16.     end while
17.     if e.PC_next == X then
18.         for all a ∈ possible_PC_next_vals(e) do
19.             e' ← e.update_PC_next(a)
20.             U.push(e')
21.             T.insert(a)
22.         end for
23.     end if
24. end while
```

input-dependent branch in depth-first order and simulation continues. When all execution paths have been simulated to the end of the application (i.e., depth-first traversal of the control flow graph terminates), activity analysis is complete.

This naïve analysis cannot terminate for applications with complex control flow or infinite loops. Figure 10 shows such an application, in which an unconditional branch jumps from the `end` basic block back to the `begin` basic block, resulting in an infinite loop. Each time the branch (`jl`) instruction on Line 6 is encountered, the `else` block will be pushed onto the stack (Lines 17 and 18 in

Algorithm 1), while simulation continues down the then block. However, since the end of the application is never reached, the stack will never be popped and symbolic simulation will never finish.

A closer look at the application in Figure 10 provides an insight. The program consists of a loop that reads an input, sets the value of either register $r4$ or $r5$, depending on the input value, then subtracts the two registers. Although the loop iterates infinitely, only two possible simulation states exist at instruction 10 (i.e., $\langle r4, r5, r6\rangle$ is either $\langle 1, 0, 1\rangle$ or $\langle 0, 1, -1\rangle$). These two states differ in only a small fraction of the processor's state (i.e., $r6$ and one bit each of $r4$ and $r5$). Thus, a *conservative state* formed by merging the two states such that differing state variables are set to unknown values (Xs) can represent both states with little loss of toggling information. After simulating a conservative state that represents both possible states, any future simulation of execution paths at Line 10 can safely be terminated since they will not identify any new toggling behavior. In this way, even an application with an infinite execution tree can be analyzed while still guaranteeing that the maximal set of gates that an application can toggle will be identified.

Algorithm 2 describes the proposed scalable symbolic coanalysis technique. The analysis initializes the symbolic simulation in the same way as Algorithm 1 (i.e., all nets and memory cells are initialized to X, the PC is loaded with the application's first address, and reset is toggled), with one addition – Algorithm 2 also initializes a *conservative* system state map. This map holds the *conservative* simulation values for each net and memory location in the design at any instruction that alters the PC (besides incrementing). Only the states at PC-altering instructions (i.e., any instruction at which the resulting PC may not be PC++, such as a branch or jump) are stored since they are the instructions that can cause path explosion due to input-dependent or infinite control structures. Each map entry's key is the PC value of the PC-altering instruction (i.e., a branch's address in program memory). The conservative value stored in the map for a state variable is assumed to be unknown (X) for any net that has been observed to have different values (i.e., 0 and 1) in a state with the same PC. Assigning a value of X to a net carries the assumption that the net can be toggled by the application for some input assignment.

Once initialization is complete, simulation begins, continuing until all paths have been explored or have been determined to be covered by a previously simulated state. During each visit to a PC-altering instruction, the current state is compared with the conservative state stored for that instruction (PC). If the current state is a substate of the stored state (i.e., the states are identical *or* the stored state has Xs in all state variables where the states differ), then all paths through the current state have already been analyzed in a previous portion of the symbolic simulation and the current execution path can be safely terminated. If the current state is not a substate of the stored state, a new conservative symbolic state is generated by assigning any nets that differ in value between the current state and the stored conservative state to Xs.[4] This new conservative state is loaded as the processor state before continuing symbolic simulation and is also stored into the conservative state map in place of the previous stored state. Symbolic simulation must continue to explore this execution path from the new conservative state because it includes new toggled gates and, therefore, the worst-case toggling activity may not have been observed yet. Symbolic simulation then continues as described by Algorithm 2.

Figure 11 shows an example of the proposed technique analyzing an application (from Figure 10) that Algorithm 1 is unable to analyze. The application contains an infinite loop of four basic blocks; the execution of two of the basic blocks is dependent on the input value read on Line 4. Algorithm 1 would attempt to explore the entire infinite execution tree (including grayed-out blocks and beyond). However, inspecting the code, it is clear that there are two possible system states at the

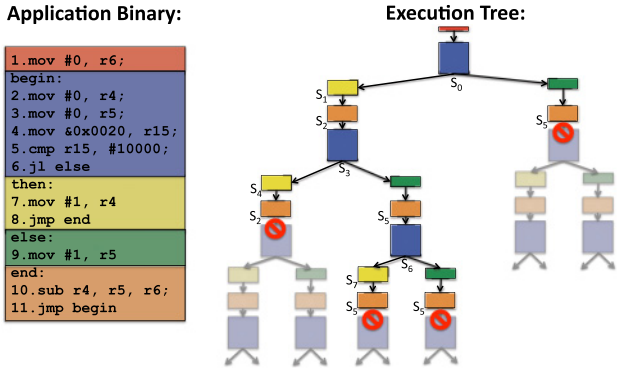
[4]The reason an X produces the worst-case toggling behavior is that even if a net has an X value in two consecutive cycles, the analysis tool considers it a possible toggle.

ALGORITHM 2: Scalable Input-Independent Gate Activity Analysis

```
1.  Procedure GateActivityAnalysis(app_binary, design_netlist)
2.  Initialize all memory cells and all gates in design_netlist to X
3.  Load app_binary into program memory
4.  Propagate reset toggle signal
5.  s ← State at start of app_binary
6.  Symbolic Execution Tree T.set_root(s)
7.  Unprocessed execution points stack, U.push(s)
8.  Conservative system state map, C.init()
9.  while U != ∅ do
10.     e ← U.pop()
11.     if e.altersPC() and e.PC ∈ C then
12.         a ← C.getState(e.PC)
13.         if e.isConservativeSubstateOf(a) then
14.             continue
15.         else
16.             e ← buildConservativeState(a, e)
17.             C ← C.update(e.PC, e)
18.         end if
19.     else
20.         C ← C.add(e.PC, e)
21.     end if
22.     while e.nextPC != X and !e.END do
23.         e.setInputsX() // set all peripheral port inputs to Xs
24.         e' ← propagateGateValues(e) // perform simulation for this cycle
25.         e.annotateGateActivity(e,e') // annotate tree point with activity
26.         e.addNextState(e') // add to execution tree
27.         e ← e' // process next cycle
28.     end while
29.     if e.nextPC == X then
30.         for all a ∈ possibleNextPCVals(e) do
31.             e' ← e.updateNextPC(a)
32.             U.push(e')
33.             T.insert(e')
34.         end for
35.     end if
36. end while
```

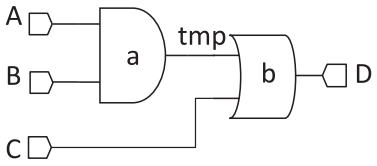
end of one iteration of the loop – $\langle r4, r5, r6\rangle$ is either $\langle 1, 0, 1\rangle$ or $\langle 0, 1, -1\rangle$. Therefore, continuing to explore the execution tree after these two states have been observed will not uncover any new toggling behavior.

During scalable symbolic coanalysis, when the conditional branch instruction at Line 6 is first reached, state S_0 is added to the conservative system state map with a key of 6 (the branch's PC). At this point, `r4`, `r5`, and `r6` are 0, while `r15` is all Xs (because its value is read from an input). As symbolic simulation continues down the `then` path, entries S_1 and S_2 are added for the branches at Lines 8 and 11, respectively. When symbolic simulation reaches the branch at Line 6 again, `r6` has a value of 1, which requires a new conservative state, S_3, to replace S_0, where `r6`'s value is 0...0X, because the least significant bit of `r6` was observed once as a 0 and once as a 1 during the branch instruction at Line 6. Continuing, state S_1 must be replaced by state S_4, because the simulation



State	PC	r4	r5	r6	r15
S_0	6	0...00	0...00	0...00	X...X
S_1	8	0...01	0...00	0...00	X...X
S_2	11	0...01	0...00	0...01	X...X
S_3	6	0...00	0...00	0...0X	X...X
S_4	8	0...01	0...00	0...0X	X...X
S_5	11	0...0X	0...0X	X...X1	X...X
S_6	6	0...00	0...00	X...XX	X...X
S_7	8	0...01	0...00	X...XX	X...X

Fig. 11. Scalable input-independent gate activity analysis can analyze applications with infinite loops and input-dependent branches by simulating conservative states that capture the activity of multiple possible states.



State	A	B	C	tmp	D
S_0	1	0	0	0	0
S_1	0	1	0	0	0
Cons S_1	X	X	0	X	X

Fig. 12. Simulating from a conservative state in our scalable input-independent gate activity analysis (Algorithm 2) may result in identifying some gates as possibly exercisable that would not be reported as such by the baseline approach (Algorithm 1). The conservative nature of this approach allows analysis of applications with complex control structures while maintaining the guarantee that all exercisable gates will be identified.

value of r6 is 0...0X during the next simulation of the branch at Line 8. Since the value of r6 is then overwritten to be 1 again prior to Line 11, this execution path observes state S_2 for a second time, indicating that no further exploration is required, and the path is terminated. Next, the last else block is popped off the execution points stack. For this path, r6 becomes -1 (i.e., 1...11), which is not a substate of the latest conservative state stored for the branch at line 11 – S_2. Thus, a new state, S_5, replaces S_2. At the end of the next iteration of the main loop, the current state is a substate of S_5;thus, exploration is halted for that path and the top of the execution points stack is popped. After this, all states encountered are substates of previously explored states stored in the conservative system state map. In this example, our scalable symbolic coanalysis technique simulates only 35 dynamic instructions rather than the infinite number that the previous coanalysis technique would attempt to simulate. The proposed scalable symbolic coanalysis technique captures the worst-case toggling activity of an application with $O(n^2 * m)$ complexity, where n is the number of basic blocks in the application and m is the number of gates in the processor's netlist.

One drawback of the scalable technique is the inaccuracy introduced by being conservative when recording toggling activity. Consider Figure 12. Assume that states S_0 and S_1 are two different states observed for the same PC, arrived at sequentially by the baseline coanalysis. In the proposed scalable technique, when simulation arrives at state S_1 after having previously observed S_0 for the same PC, a new conservative state, ConsS_1, will be created and stored as the current conservative state for this PC. ConsS_1 is then used to continue symbolic simulation instead of S_1. Therefore, the scalable approach must conservatively assume that both gates a and b are toggling when the

baseline coanalysis may show them to not toggle. Since usually only a small number of state values differ between executions of the same static instruction, the difference between the scalable and baseline symbolic coanalysis approaches is expected to be small. However, the inaccuracy introduced by our scalable technique will only make analysis more conservative. This means that the peak power bounds derived from this scalable coanalysis may be somewhat higher than the naïve approach but will always represent a guaranteed bound.[5]

During symbolic simulation, the simulator captures the activity of each gate at each point in the execution tree. A gate is considered active if its value changes or if it has an unknown value (X) and is driven by an active gate; otherwise, the gate is idle. The resulting annotated symbolic execution tree describes all possible instances in which a gate could possibly toggle for all possible executions of the application binary. As such, a gate that is not marked as toggled at a particular location in the execution tree can *never* toggle at that location in the application. As described in the next sections, we can use the information gathered during activity analysis to bound the peak power and energy requirements of an application.

3.2 Input-Independent Peak Power Requirements

The input to the second phase of our technique is the symbolic execution tree generated by input-independent gate activity analysis. Algorithm 3 describes how to use the activity-annotated execution tree to generate peak power requirements for a ULP processor, application pair.

The first step in determining peak power from an execution tree produced during gate activity analysis is to concatenate the execution paths in the execution tree into a single execution trace. We use a value change dump (VCD) file to record the gate-level activity in the execution trace. The execution trace contains Xs; the goal of the peak power computation is to assign values to the Xs in a way that maximizes power for each cycle in the execution trace. The power of a gate in a particular cycle is maximized when the gate transitions (toggles). Since a transition involves two cycles, maximizing dynamic power in a particular cycle, c, of the execution trace involves assigning values to any Xs in the activity profiles of the current and previous cycles, c and $c-1$, to maximize the number of transitions in cycle c.

The number and power of transitions are maximized as follows. When the output value of a gate in only one of the cycles, c or $c-1$, is an X, the X is assigned the value that assumes that a transition happened in cycle c. When both values are Xs, the values are assigned to produce the transition that maximizes power in cycle c. The maximum power transition is found by a look-up into the standard cell library for the gate. Since constraining Xs in two consecutive cycles to maximize power in the second cycle may not maximize power in the first cycle, we produce two separate VCD files – one that maximizes power in all even cycles and one the maximizes power in all odd cycles. To find the peak power of the application, we first run activity-based power analysis on the design using the even and odd VCD files to generate even and odd power traces. We then form a peak power trace by interleaving the power values from the even cycles in the even power trace and the odd cycles in the odd power trace. This peak power trace bounds the peak power that is possible in every cycle of the execution trace. The peak power requirement of the application is the maximum per-cycle power value found in the peak power trace.[6]

[5]Four of our benchmarks (`div`, `inSort`, `rle`, and `Viterbi`) require this scalable technique in order to be analyzed in a tractable amount of time. Other benchmarks see significant reductions in analysis time (e.g., `binSearch`'s analysis time is reduced by 97%). The conservative inaccuracy results in <2% increase in gate toggling for those benchmarks that are tractable for the naïve analysis. Despite this inaccuracy, Section 5 shows that this scalable technique can provide significantly tighter peak power and energy bounds than application-agnostic approaches.

[6]It is possible that glitching between clock edges can impact the power profile for an application. This impact can be accounted for by Primetime's power analysis (Synopsys 2015).

	1	2	3	4	5	6	7	8	9
g1	0	0	1	X	X	X	0	0	0
g2	0	X	X	X	X	X	X	0	0
g3	0	0	0	1	X	X	X	X	0

	1	2	3	4	5	6	7	8	9
g1	0	0	1	0	0	1	1	0	0
g2	0	1	0	1	0	1	1	0	0
g3	0	0	0	1	0	1	0	1	0

	1	2	3	4	5	6	7	8	9
g1	0	0	1	0	1	1	0	0	0
g2	0	0	1	0	1	0	1	0	0
g3	0	0	0	1	0	0	1	1	0

Fig. 13. To determine a bound on peak power, we generate two different activity profiles – one that maximizes power in even cycles (left) and one that maximizes power in odd cycles (right).

ALGORITHM 3: Input-Independent Peak Power Computation

1. **Procedure** *Calculate Peak Power*
2. {E|O}_VCD ← Open {Even|Odd} VCD File // maximizes peak power in even|odd cycles
3. T ← flatten(Execution Tree) // create a flattened execution trace that represents the execution tree
4. **for all** {even|odd} cycles c ∈ T **do**
5. **for all** toggled gates g ∈ c **do**
6. **if** value(g,c) == X && value(g,c-1) == X **then**
7. value(g,c-1) ← maxTransition(g,1) // returns the value of the gate in the first cycle of the gate's maximum power transition
8. value(g,c) ← maxTransition(g,2) // returns the value of the gate in the second cycle of the gate's maximum power transition
9. **else if** value(g,c) == X **then**
10. value(g,c) ← !value(g,c-1)
11. **else if** value(g,c-1) == X **then**
12. value(g,c-1) ← !value(g,c)
13. **end if**
14. **end for**
15. {E|O}_VCD ← value(*,c-1)
16. {E|O}_VCD ← value(*,c)
17. **end for**
18. Perform power analysis using E_VCD and O_VCD to generate even and odd power traces, P_E and P_O
19. Interleave even cycle power from P_E with odd cycle power from P_O to form peak power trace, P_{peak}
20. peak power ← max(P_{peak})

Our VCD generation technique is illustrated in Figure 13. We use the example of three gates with overlapping Xs that need to be assigned to maximize power in every cycle. We show two assignments – one that maximizes peak power in all even cycles (left) and one that maximizes peak power in all odd cycles (right). Assuming, for the sake of example, that all gates have equal power consumption and that the 0 → 1 transition consumes more power than the 1 → 0 transition

for these gates, the highest possible peak power for this example happens in cycle 6 in the "even" activity trace, when all the gates have a $0 \rightarrow 1$ transition.

3.3 Input-Independent Peak Energy Requirements

Our technique generates a per-cycle peak power trace characterizing all possible execution paths of an application. The peak power trace can be used to generate peak energy requirements. Figure 14 shows per-cycle peak power traces sampled from our benchmark applications. Since per-cycle peak power varies significantly over the compute phases of an application, peak energy can be significantly lower than assuming the maximum peak energy (i.e., $peak\ power * clock\ period * number\ of\ cycles$). Instead, the peak energy of an application is bounded by the execution path with the highest sum of per-cycle peak power multiplied by the clock period. To avoid enumerating all execution paths, we use several techniques. For an input-dependent branch, peak energy is computed by selecting the branch path with higher energy. For a loop whose number of iterations is input independent, peak energy can be computed as the peak energy of one iteration multiplied by the number of iterations. For cases in which the number of iterations is input dependent, the maximum number of iterations may be determined either by static analysis or user input (as suggested by prior work (Jayaseelan et al. 2006)).[7] If neither is possible, it may not be possible to compute the peak energy of the application; however, this is uncommon in embedded applications. In fact, all 43 benchmarks we analyzed in Embedded Sensors (Zhai et al. 2009), EEMBC (2017), and MiBench (Guthaus et al. 2001) suites have bounded execution time.

3.4 Validation of X-based Analysis

To demonstrate that our symbolic execution-based (X-based) activity analysis marks all gates that could possibly be toggled by an application for all possible inputs, we performed a validation check by comparing the sets of gates toggled by input-based simulations for several different input sets against the set of gates marked as potentially toggled by symbolic simulation. Figure 15 illustrates this comparison for two input-based simulations of the *mult* benchmark with different input sets – those that have the lowest and highest number of toggled gates. In the figure, toggled gates common to X-based and input-based simulation are shown as Xs and gates that are exclusively marked by symbolic simulation as potentially toggled are shown as blue triangles. As expected, there are no gates that are exclusively marked by input-based simulation. Our validation results show that all the gates toggled by input-based simulation are also marked as potentially toggled by X-based symbolic simulation, validating the correctness of our approach for characterizing toggle activity.

We perform a second validation of our technique by comparing the peak power traces generated for benchmarks by our technique against power traces generated by input-based execution of the benchmarks. The validation results confirm that our peak power trace always provides an upper bound on the power of any input-based power trace. Figure 16 shows an example; the X-based peak power trace for the *mult* application is always higher than the input-based power trace. These validation results also show that the X-based peak power trace closely matches the input-based trace, indicating that the peak power and energy requirements generated by our technique are not overly conservative.

3.5 Enabling Peak Power Optimizations

Since our technique is able to associate the input-independent peak power consumption of a processor with the particular instructions that are in the pipeline during a spike in peak power, we can

[7] The number of loop iterations is bounded for all evaluated benchmarks. In general, applications with unbounded runtimes are uncommon in embedded domains.

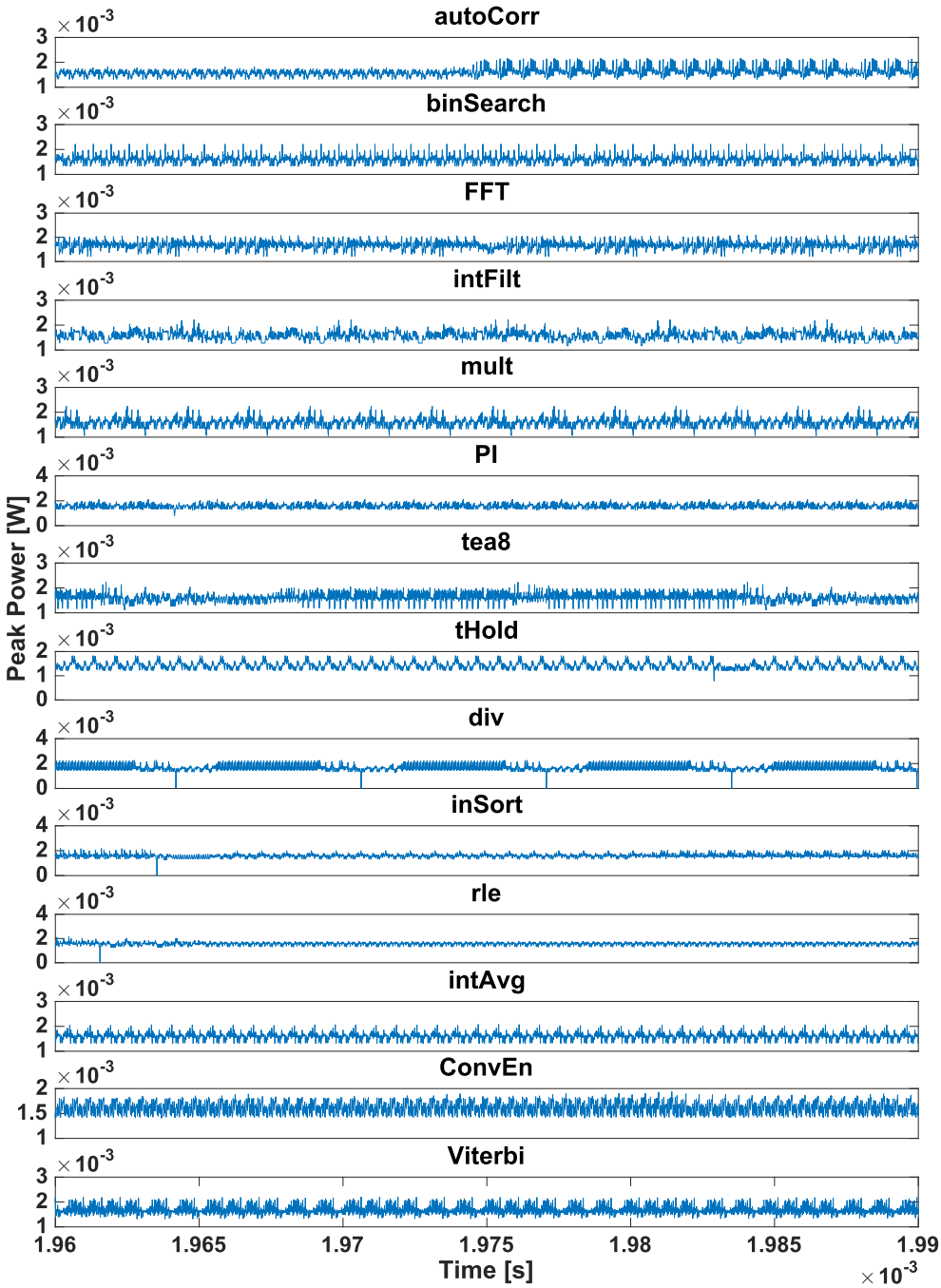


Fig. 14. The per-cycle peak power varies significantly over the course of an application, showing that the worst-case average power can be significantly lower than peak power. Therefore, the peak energy can be significantly lower than the product of peak power and application runtime would suggest.

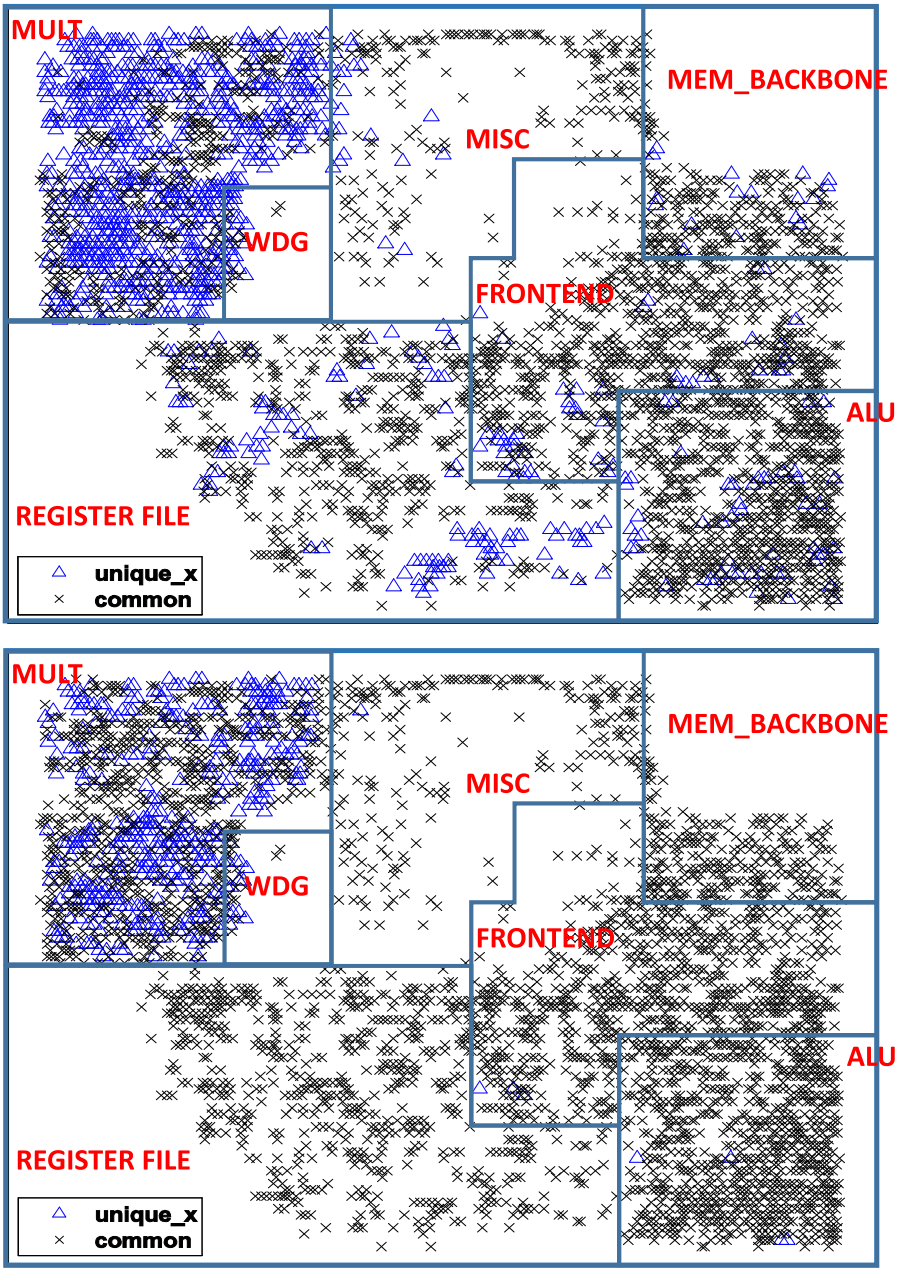


Fig. 15. Toggled gates for *mult* with low-activity inputs (top) and high-activity inputs (bottom) compared against potentially toggled gates identified by X-based analysis. X-based simulation marks all gates that can potentially toggle for an application for all possible inputs. This set of gates (unique_x ∪ common) is a superset of the gates that toggle during an input-based application execution (common).

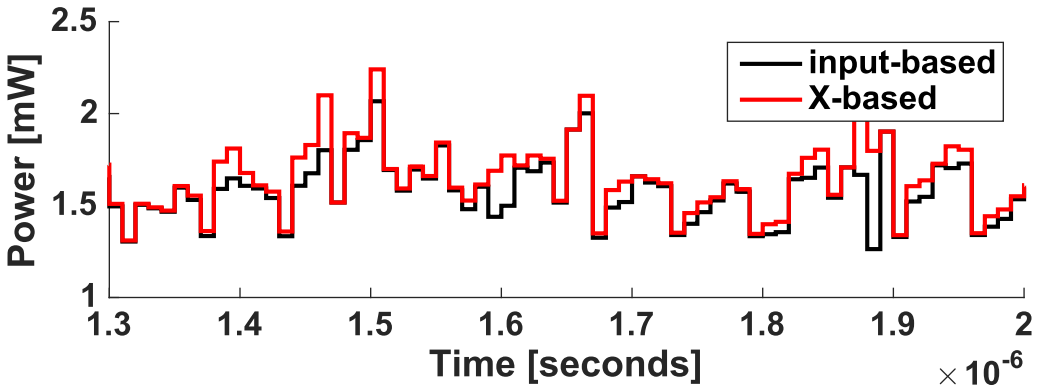


Fig. 16. The X-based peak power trace generated by our technique for an application provides an upper bound on all possible input-based power traces for the application (result shown for *mult*).

use our tool to identify which instructions or instruction sequences cause spikes in peak power. Our technique can also provide a power breakdown that shows the power consumption of the microarchitectural modules that are exercised by the instructions. These analyses can be combined to identify which instructions executing in which modules cause power spikes. After identifying the cause of a spike, we can use software optimizations to target the instruction sequences that cause peaks and replace them with alternative sequences that generates less instantaneous activity and power while maintaining the same functionality. After optimizing software to reduce a spike in peak power, we can rerun our peak power analysis technique to determine the impact of optimizations on peak power. Guided by our technique, we can choose to apply only the optimizations that are guaranteed to reduce peak power.

Figure 17 shows an example in which our technique identifies peak power spikes in cycles 146 and 150. Our technique also reports the instructions in each stage of the pipeline during those cycles of interest (COIs) and the per-module power breakdown for those cycles, which identifies the modules that are consuming the most power. This information can be used to guide optimizations that replace the instructions with different instruction sequences that induce less activity and power in the modules that consume the most power. Since software optimizations can impact performance as well as peak power, we will discuss optimizations that reduce peak power and their impact on performance and energy in Section 5.1.

4 METHODOLOGY

4.1 Simulation Infrastructure and Benchmarks

We verify our technique on a silicon-proven processor – openMSP430 (Girard 2013), an open-source version of one of the most popular ULP processors (Borgeson 2012; Wikipedia 2016). The processor is synthesized, placed, and routed in TSMC 65GP technology (65nm) for an operating point of 1V and 100MHz using Synopsys Design Compiler (Synopsys 2015) and Cadence EDI System (Cadence 2014). Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using a custom gate-level simulator that efficiently traverses the control flow graph of an application and captures input-independent activity profiles (Section 3). We show results for all benchmarks from Zhai et al. (2009) and all EEMBC benchmarks that fit in the program memory of the processor. These benchmarks are chosen to be representative of emerging ultra-low-power application domains such as wearables, the Internet of Things, and

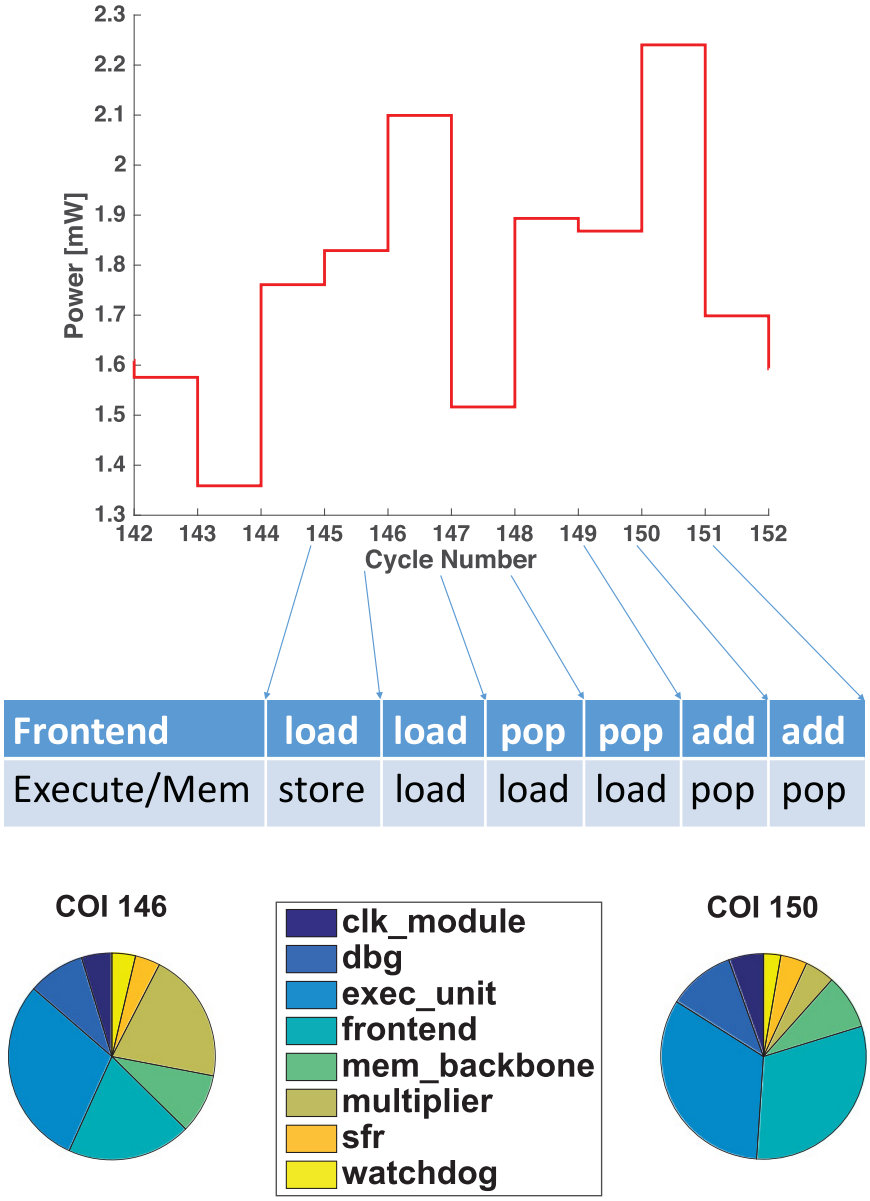
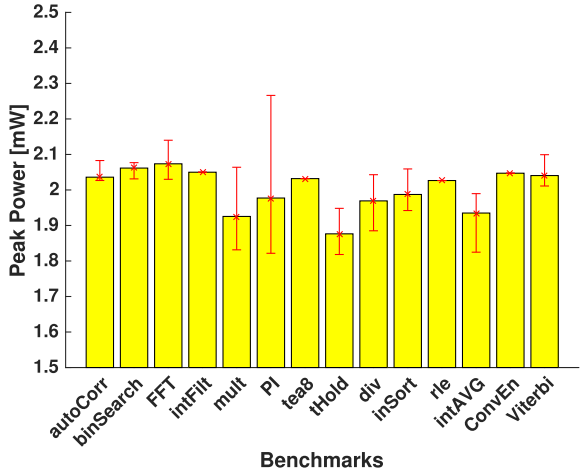
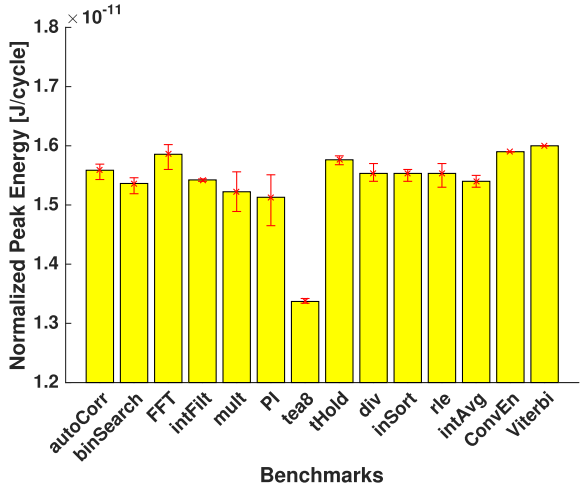


Fig. 17. A snapshot of instantaneous power profiles for *mult* at two different COIs where peaks occur. Our technique analyzes the instructions in the pipeline (top) to find each COI's culprit instructions that cause the peak power in each pipeline stage along with the per-module peak power breakdown (bottom) to identify which instructions in which microarchitectural modules are responsible for a peak.

sensor networks (Zhai et al. 2009). The IPC of these benchmarks on our processor varies from 1.25 to 1.39, with an average of 1.29. Power analysis is performed using Synopsys Primetime (Synopsys 2015). Experiments were performed on a server housing two Intel Xeon E-2640 processors (8 cores each, 2GHz operating frequency, 64GB RAM).



(a) Peak Power



(b) NPE

Fig. 18. Different applications and different input sets for the same application have different peak power and peak energy requirements (results for openMSP430).

Section 2 shows measured data for an MSP430F1610 processor demonstrating that different applications have different peak power and energy requirements and that the requirements of an application can vary significantly for different inputs. The results motivate an application-specific, input-independent technique for determining the peak power and energy requirements for ULP processors. For the results in Section 5, we perform evaluations on the open source openMSP430 processor (Girard 2013). Figures 18(a) and 18(b) confirm that the peak power and energy requirements of openMSP430 also depend on the application and application inputs. Note that the results in Figure 7 and Figure 18 differ because they are for different implementations of the MSP430 architecture (MSP430F1610 and openMSP430), with different process technology (130nm vs. 65nm) and operating frequencies (8MHz vs. 100MHz).

Table 3. Benchmarks

Embedded Sensor Benchmarks (Zhai et al. 2009)
mult, binSearch, tea8, intFilt,
tHold, div, inSort, rle, intAVG
EEMBC Embedded Benchmarks (EEMBC 2017)
Autocorr, FFT, ConvEn, Viterbi
Control Systems Benchmark
Proportional Integral Controller (PI)

4.2 Baselines

For baselines, we compare against conventional techniques for determining the peak power and energy requirements of processors. An overview of the baseline techniques can be found in Figure 4. The design specification-based baseline (*design tool*) is determined by performing power and energy analysis of the design using the default input toggle rate used by our design tools (Synopsys 2015). The stressmark-based baselines (*GB input-based*) use stressmarks that target peak instantaneous power and average power. Kim et al. (2012) used a genetic algorithm to automatically generate stressmarks that target maximum di/dt-induced voltage droop for a microprocessor. We modified their framework to generate stressmarks that target peak instantaneous power and average power for openMSP430. The profiling-based baseline (*input-based*) is generated by performing input-based power and energy profiling for several input sets and applying a guardbanding factor of 4/3 to the peak power and energy observed during profiling. The guardbanding factor is the same as in prior studies (Intel Corporation 2000; Kontorinis et al. 2009) and is appropriate for the input-dependent peak power variability exhibited by our benchmarks (Figure 7(a)).

5 RESULTS

We use our technique described in Section 3 to determine peak power and energy requirements for a ULP processor for different benchmark applications. Figure 19 compares the peak power requirements reported by our technique against the conventional techniques for determining peak power requirements, described in Section 4.2. The results show that the peak power requirements reported by our X-based technique are higher than the highest input-based, application-specific peak power for all applications, confirming that our technique provides a bound on peak power. The results also show that our technique provides the most accurate bound on peak power compared to conventional techniques for determining peak power requirements. For example, the peak power requirements reported by our technique are only 1% higher than the highest observed input-based peak power for the benchmark applications, on average. Other techniques for determining peak power and energy requirements are significantly less accurate, which can lead to inefficiency in critical system parameters, such as size and weight (see Section 1).

Our technique is more accurate than application-oblivious techniques, such as determining peak power requirements from a stressmark or design specification, because an application constrains which parts of the processor can be exercised in a particular cycle. Our technique also provides a more accurate bound than a guardbanded input-based peak power requirement because it does not require a guardband to account for the nondeterminism of input-based profiling (shown in Figure 19 as error bars). By accounting for all possible inputs using symbolic simulation, our technique can bound peak power and energy for all possible application executions without guardbanding. The peak power requirements reported by our technique are 15% lower than guardbanded application-specific requirements, 26% lower than guardbanded stressmark-based requirements, and 27% lower than design specification-based requirements, on average.

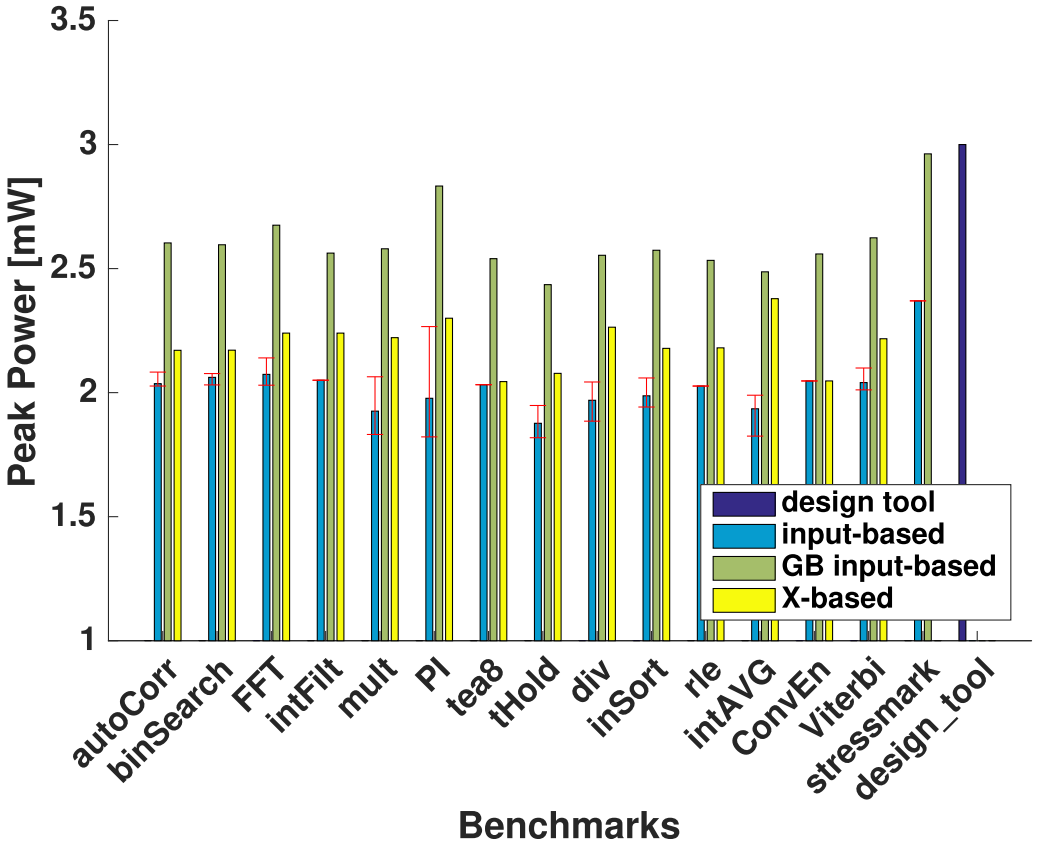


Fig. 19. Our X-based technique for determining peak power requirements provides the most accurate (least conservative) guaranteed bound on peak power.

Since our technique is application specific and does not require guardbands, one question is, "Why is the bound provided by X-based analysis more conservative for some applications than others?" The answer is that since X-based analysis provides a bound on power for all possible inputs, it becomes more conservative when there is greater possibility for input-dependent variation in power. For example, the multiplier is a relatively large, high-power module, with high potential for input-dependent variation in power consumption. For some inputs (e.g., $X * 0$), power consumed by the multiplier is minimal since there are no partial products to compute. For other inputs (e.g., two very large numbers), the power consumed by the multiplier is much larger. Since our symbolic simulation technique assumes Xs for inputs, we always assume the highest possible power for a multiple instruction. Therefore, X-based peak power requirements for applications that contain a large number of multiplications may be more conservative than X-based requirements for other applications.

Conversely, the tea8 application, which performs encryption, uses only low-power ALU modules – shift register and XOR – that have significantly less potential for input-induced power variation. As a result, X-based analysis closely matches input-based profiling results for this application. For all applications, even those with more potential for input-induced power variation, our X-based analysis technique provides a peak power bound that is more accurate than those provided by conventional techniques.

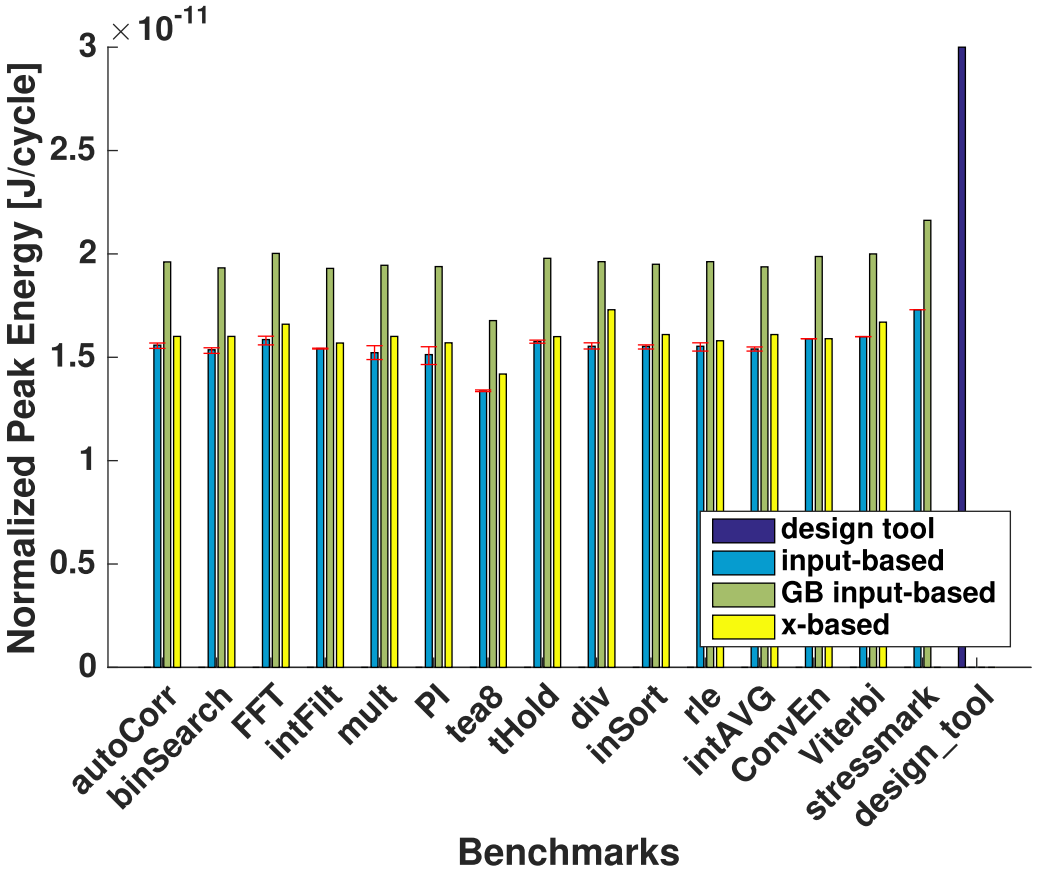


Fig. 20. Our X-based technique for determining peak energy requirement (normalized to application runtime in cycles, i.e., the peak average power) is more accurate than existing conventional techniques.

Our technique also provides more accurate bounds on peak energy than conventional techniques, partly because of the reasons mentioned above and because our technique is able to characterize the peak energy consumption in each cycle of execution, generating a peak energy trace that accounts for dynamic variations in energy consumption. Using a design specification to determine peak energy is particularly inaccurate since it does not consider dynamic variations in the energy requirements of an application. The guardbanded input-based technique, which does consider dynamic variations, provides a more accurate peak energy bound than the design specification for all benchmarks. However, it does not always provide a more accurate bound than the design specification for peak power since peak power is an instantaneous phenomenon that is less dependent on dynamic variations. Figure 20 presents peak energy of different benchmarks normalized to application runtime in cycles, i.e., peak average power, which characterizes the maximum rate at which the application can consume energy. In Figure 20, the peak energy requirements reported by our technique are 17% lower than guardbanded application-specific requirements, 26% lower than guardbanded stressmark-based requirements, and 47% lower than design specification-based requirements, on average. As expected, application-specific normalized peak energy (Figure 20) varies less than peak power (Figure 19) since peak energy characterizes average peak power over the entire

Table 4. Percentage Reduction in Harvester Area Compared to Different Baseline Techniques, Averaged Over All Benchmarks, for Different Percentage Contributions of the Processor Peak Power to the System Peak Power

Baseline	10%	25%	50%	75%	90%	100%
GB-Input	1.49	3.73	7.47	11.21	13.45	14.94
GB-Stress	2.60	6.47	12.95	19.42	23.31	25.90
Design Tool	2.68	6.70	13.41	20.12	24.14	26.82

Table 5. Percentage Reduction in Battery Volume Compared to Different Baseline Techniques, Averaged Over All Benchmarks, for Different Percentage Contributions of the Processor Energy to the Overall Energy of the System

Baseline	10%	25%	50%	75%	90%	100%
GB-Input	1.74	4.37	8.74	13.11	15.73	17.48
GB-Stress	2.59	6.49	12.98	19.48	23.37	25.97
Design Tool	4.66	11.66	23.32	34.98	41.97	46.64

execution of an application, whereas peak power corresponds to one instant in the application's execution.

As described in Section 1, more accurate peak power and energy requirements can be leveraged to reduce critical ULP system parameters, such as size and weight. For example, reduction in a Type 1 system's peak power requirements allows a smaller energy harvester to be used. System size is roughly proportional to harvester size in Type 1 systems. In Type 2 systems, it is the peak energy requirement that determines the harvester size; reduction in peak energy requirement reduces system size roughly proportionally. Since required battery capacity depends on a system's peak energy requirement and effective battery capacity depends on the peak power requirement, reductions in peak power and energy requirements both reduce battery size for Type 2 and Type 3 systems.

A ULP system may contain other components, such as transmitter/receiver, ADC, DAC, and sensor(s), along with the processor. All of these components may contribute to the system's peak power and energy and, hence, the sizing of the harvester and battery. Tables 4 and 5 show the percentage reduction in the harvester size and battery size, respectively, from our technique for different fractions representing the processor's contribution to the system's peak power and energy. For a real system such as the one shown in Figure 2, which has a harvester area of $32.6cm^2$ and a battery volume of $6.95mm^3$, the area reduction of the harvester is 4.87, 8.44, or $8.75cm^2$ if the system is designed using guardbanded input-based profiling, guardbanded stressmark, or design tool, respectively, for estimating the peak power of the processor. Similarly, the volume reduction of the battery is 0.42, 0.63, or $1.12mm^3$, respectively.[8] As expected, savings from our technique are higher when the processor is the dominant consumer of power and energy in the overall system.[9]

The reduction in battery capacity allowed by our tighter peak energy bound can have a direct cost, weight, and size savings when specific components of a system are considered. An analysis

[8]The battery is a thin film battery of dimensions 5.7mm × 6.1mm × 200μm (area of $34.7mm^2$). Assuming that the height of the battery does not change, the corresponding savings in battery area are 6.07, 9.01, and $16.18mm^2$, respectively.
[9]ITRS 2015 projections show that the microcontroller will be the dominant consumer of power in future IoT and IoE systems (ITRS 2015).

of relevant batteries from Digikey suggests that if peak energy requirements of a Type 3 system (Figure 1) can be reduced by 10%, a switch can be made from a 50mAh lithium battery to a 45mAh lithium battery, which, in turn, can result in a size reduction of 13% (23mm, coin to 20mm, coin), a weight reduction of 37% (3.5g to 2.2g), and a cost reduction of 74% ($7.80 to $1.90). As another example, a 15% reduction in peak energy requirements can allow a switch from 6mAh lithium battery to a 5.8mAh lithium battery, which, in turn, leads to a size reduction of 25% (9.5mm $\times$ 2.1mm to 6.8mm $\times$ 2.2mm), a weight reduction of 50% (0.45g to 0.227g), and a cost reduction of 34% ($3.30 to $2.19). A modest size reduction in battery requirements can result in a large reduction in size and cost of the components of a ULP system. Since the battery and/or harvester are often among the largest and most expensive components in a ULP system, this can translate into significant systems cost savings, which are especially critical in the volume market that ULP systems often target. Similarly, saving the area of the harvester can also reduce the area of the entire system. For example, we can reduce the PCB area of the system in Figure 2 by 9%, 15%, and 16% over the baselines of GB input, GB stress, and Design Tool, respectively, for a Type 1 system.

5.1 Optimizations

As discussed in Section 3.5, our technique can be used to guide application-level optimizations that reduce peak power. Here, we discuss three software optimizations suggested by our technique that we applied to the benchmark applications to reduce peak power. The optimizations were derived by analyzing the processor's behavior during the cycles of peak power consumption. This analysis involves (a) identifying instructions in the pipeline at the peak and (b) identifying the power contributions of the microarchitectural modules to the peak power to determine which modules contribute the most.

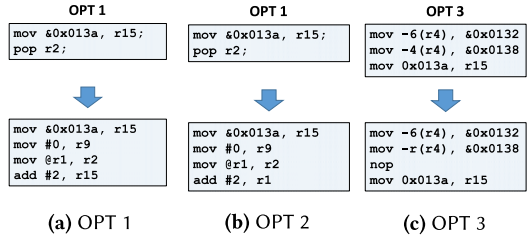
The first optimization aims to reduce a peak by "spreading out" the power consumed in a peak cycle over multiple cycles. This is accomplished by replacing a complex instruction that induces a lot of activity in one cycle with a sequence of simpler instructions that spread the activity out over several cycles.

The second optimization aims to reduce the instantaneous activity in a peak cycle by delaying the activation of one or more modules, previously activated in a peak cycle, until a later cycle. For this optimization, we focus on the `POP` instruction, since it generates peaks in some benchmarks. The peaks are caused by a `POP` instruction generating high activity on the data and address buses and simultaneously using the incrementer logic to update the stack pointer. To reduce the peak, we break down the `POP` instruction into two instructions – one that moves data from the stack and one that increments the stack pointer.

The third optimization is based on the observation that, for some applications, peak power is caused by the multiplier (a high-power peripheral module) being active simultaneously with the processor core. To reduce peak power in such scenarios, we insert a `NOP` into the pipeline during the cycle in which the multiplier is active.

The three optimizations we applied to our benchmarks to reduce peak power are summarized below. The optimizations are shown in Figure 21.

- **Register-Indexed Loads (OPT 1):** A load instruction (`MOV`) that references the memory by computing the address as an offset to a register's value involves several micro-operations – source address generation, source read, and execute. Breaking the micro-operations into separate instructions can reduce the instantaneous power of the load instruction. The ISA already provides a register indirect load operation in which the value of the register is directly used as the memory address instead of as an offset. Using another instruction (such



(a) OPT 1 **(b)** OPT 2 **(c)** OPT 3

Fig. 21. Instruction optimization transforms.

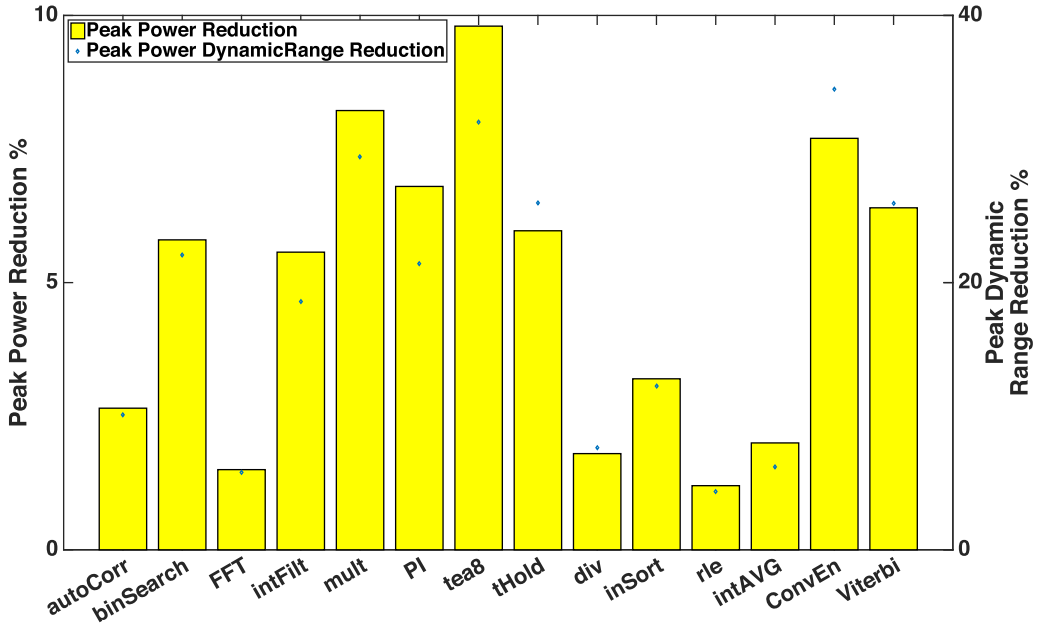


Fig. 22. Peak power reduction (left axis) and peak power dynamic range reduction (right axis) achieved by optimizations. These reductions are enabled by our analysis tool and provide further reduction in energy harvester size.

as an ADD or SUB), we can compute the correct address and store it into another register. We then use the second register to execute the load in register-indirect mode.

- **POP instructions (OPT 2):** The micro-operations of a POP instruction are (a) read value from address pointed to by the stack pointer and (b) increment the stack pointer by two. POP is emulated using `MOV @SP+, dst`. This can be broken down to two instructions – `MOV @SP, dst` and `ADD #2, SP`.
- **Multiply (OPT 3):** The multiplier is a peripheral in openMSP430. Data is MOVed to the inputs of the multiplier and then the output is MOVed back to the processor. For a 2-cycle multiplier, all moving of data can be done consecutively without any waiting. However, this involves a high power draw, since there will be a cycle when both the multiplier and the processor are active. This can be avoided by adding a NOP between writing to and reading from the multiplier.

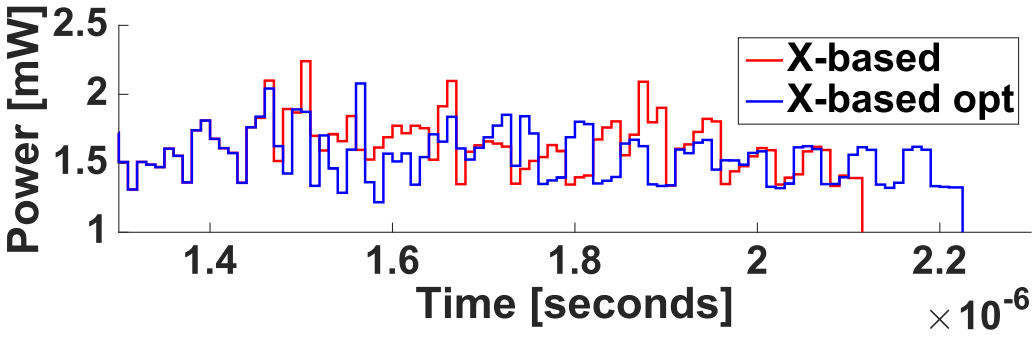


Fig. 23. A snapshot of instantaneous power profiles for mult before and after optimization.

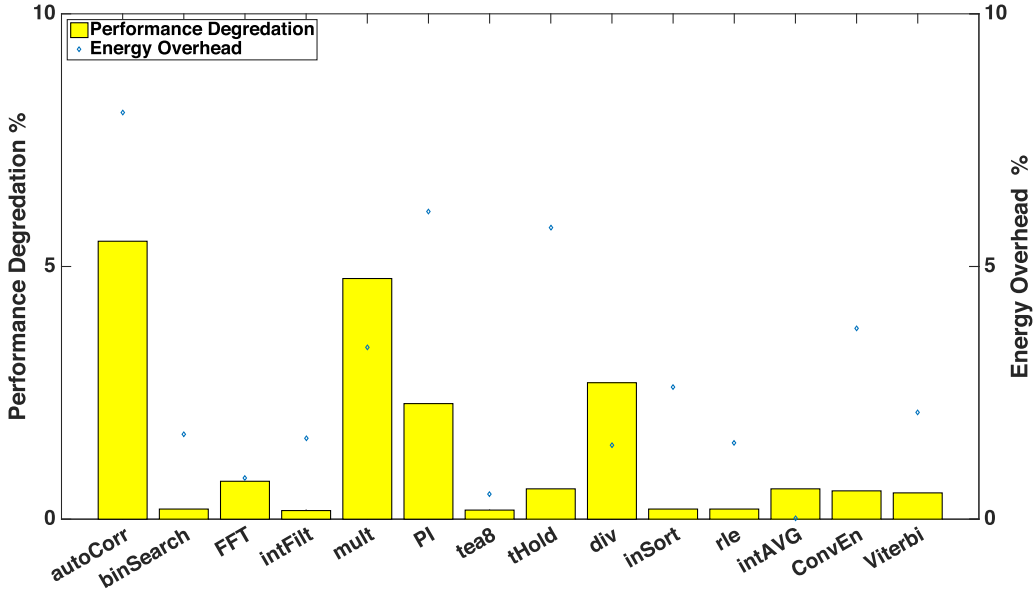


Fig. 24. Performance degradation and energy overhead introduced by peak power optimizations is small (average: 1%).

Figure 22 shows the reduction in peak power achieved by applying the optimizations motivated by our technique. Results are quantified in terms of peak power reduction, as well as reduction in peak power dynamic range, which quantifies the difference between peak and average power. Peak power dynamic range decreases as peaks are reduced closer to the range of average power. Reduction in peak power dynamic range can improve battery lifetime in Type 2 and Type 3 systems, and reduction in peak power requirements can be leveraged to reduce harvester size in Type 1 systems (see Section 1). Our results show that peak power can be reduced by up to 10%, and 5% on average. Peak power dynamic range can be reduced by up to 34%, and 18% on average. Figure 23 shows the peak power traces for an example application before and after optimization, demonstrating that optimization can reduce the peak power requirements for an application.

Since optimizations that reduce peak power can increase the number of instructions executed by an application, we evaluated the performance and energy impact of the optimizations. Figure 24

Table 6. Microarchitectural Features in Recent Embedded Processors

Processor	Branch Predictor	Cache
ARM Cortex-M0	no	no
ARM Cortex-M3	yes	no
Atmel ATxmega128A4	no	no
Freescale/NXP MC13224v	no	no
Intel Quark-D1000	yes	yes
Jennic/NXP JN5169	no	no
SiLab Si2012	no	no
TI MSP430	no	no

shows the results. Applying the optimizations suggested by our technique degrades performance by up to 5% for one application and by 1%, on average. On average, the optimizations increase energy by 3%. Although the optimizations increase energy slightly, they can still enable reduction in size for Type 1 systems, in which harvester size is dictated by peak power, and may also reduce the size of Type 2 and Type 3 systems, in which both peak power and energy determine the size of energy storage and harvesting components (see Figure 3).

6 GENERALITY AND LIMITATIONS

We applied our techniques in the context of ULP processors that are already the most widely used type of processor and are also expected to power a large number of emerging applications (Dunkels et al. 2012; Magno et al. 2013; Park et al. 2006; Tessier et al. 2005; Yu and Watteyne 2013). Such processors also tend to be simple, run relatively simple applications, and do not support nondeterminism (no branch prediction and caching; for example, see Table 6). This makes our symbolic simulation-based technique a good fit for such processors. Below, we discuss how our technique may scale for complex processors and applications, if necessary.

More complex processors contain more performance-enhancing features – such as large caches, prediction or speculation mechanisms, and out-of-order execution – that introduce nondeterminism into the instruction stream. Coanalysis is capable of handling this added nondeterminism at the expense of analysis tool runtime. For example, by injecting an X as the result of a tag check, both the cache hit and miss paths will be explored in the memory hierarchy. Similarly, since coanalysis already explores taken and not-taken paths for input-dependent branches, it can be adapted to handle branch prediction. In an out-of-order processor, the ordering of instructions is based on the dependence pattern between instructions. Thus, extending input-independent CFG exploration to also explore the data flow graph (DFG) may allow analysis of out-of-order execution.

In other application domains, there exist applications with more complex CFGs. For more complex applications, heuristic techniques may be used to improve scalability of hardware–software coanalysis. While heuristics have been applied to improve scalability in other contexts (e.g., verification) (Cadar and Sen 2013; Hamaguchi 2001), heuristics for hardware–software coanalysis must be conservative to guarantee that no gate is marked as untoggled when it could be toggled. The development of such heuristics is the subject of future work.

In a multiprogrammed setting (including systems that support dynamic linking), we take the union of the toggle activities of all applications (caller, callee, and the relevant OS code in the case of dynamic linking) to get a conservative peak power value. For self-modifying code, peak power for the processor would be chosen to be the peak of the code version with the highest peak.

In the case of fine-grained multithreading, any state that is not maintained as part of a thread's context is assumed to have a value of X when symbolic execution is performed for an instruction belonging to the thread. This leads to a safe guarantee of peak power for the thread irrespective of the behavior of the other threads.

Our technique naturally handles state machines that run synchronously with the microcontroller. For state machines that run asynchronously (e.g., ADCs, DACs, bus controllers), we assume the worst-case power at any instant by separately analyzing the asynchronous state machine to compute peak power and energy and adding the values to those of the processor. Asynchronous state machines are generally much smaller than the actual processor, allowing us to not be overly conservative.

A similar approach can be used to handle interrupts, i.e., offset the peak power with the worst power consumed during interrupt detection. The effect of an asynchronous interrupt can be characterized by forcing the interrupt pin to always read an X. Since this can potentially cause the PC to be updated with an X, we can force the PC update logic to ignore the interrupt handling logic's output. This is achieved by monitoring a particular net in the design and forcing it to zero every time its value becomes X. Interrupt service routines (ISRs) are regular software routines and can be analyzed with the rest of the code.

7 RELATED WORK

Peak power has been analyzed in several settings in literature. In particular, several techniques have been proposed to estimate the peak power of a design. Hsiao (1999) and Hsiao et al. (1997) propose a genetic algorithm-based estimation of peak power for a circuit. Wang and Roy (1998) use an automatic test generation technique to compute lower and upper bounds for maximum power dissipation for a VLSI circuit. Sambamurthy et al. (2009) propose a technique that uses a bounded model checker to estimate peak dynamic power at the module level. The technique is also functionally valid at the processor level. Najeeb et al. (2007) propose a technique that converts a circuit behavioral model to an integer constraint model and employs an integer constraint solver to generate a power virus that can be used to estimate the peak power of the processor. To the best of our knowledge, no prior work exists on determining application-specific peak power for a processor based on symbolic simulation.

The above techniques require a low-level description of the processor (behavioral or gate level). Techniques have also been proposed at the architecture level to predict when power exceeds the peak power budget or to lower the peak-to-average power variation. Sartori and Kumar (2009) propose the use of DVFS techniques to manage peak power in a multicore system. Kontorinis et al. (2009) proposed a configurable core to meet peak power constraints with minimal impact on performance. Our technique identifies the peak power and energy requirements of a processor through hardware–software coanalysis.

Estimating peak energy of an application has been previously studied as the worst-case energy consumption (WCEC) problem (Jayaseelan et al. 2006; Seth et al. 2006; Wägemann et al. 2015). However, prior techniques do not use accurate power models, instead relying on microarchitectural models, which do not consider the detailed state of a processor or input values. As observed by Morse et al. (2016), the power of an instruction can differ based on the previous instructions in the pipeline and its operand values. Our peak power computation technique analyzes an application on a gate-level processor netlist, allowing us to account for the fine-grained interaction between instructions and the worst-case operand values. The result is an accurate power model that can be used for WCEC analyses such as the example analysis in Section 5. Prior work on worst-case timing analysis simply identified the timing-critical path through the program.

However, the timing-critical path through a program may not be energy critical (Jayaseelan et al. 2006; Seth et al. 2006). We calculate energy across all paths through gate-level simulation to determine the path with highest energy.

Symbolic simulation has been applied in circuits for logic and timing verification as well as sequential test generation (Bryant 1991; Feng et al. 2003; Jain and Gopalakrishnan 1994; Kolbi et al. 2001; Liu and Vasudevan 2011) and determination of application-specific V_{min} (Cherupalli et al. 2016). Symbolic simulation has also been applied for software verification (Zhang et al. 2012). However, to the best of our knowledge, no existing technique has applied symbolic simulation to determine the peak power and energy requirements of an application running on a processor.

8 CONCLUSION

In this article, we showed that peak power and energy requirements for an ultra-low-power embedded processor can be application specific as well as input specific. This renders profiling methods to determine the peak power and energy of ULP processors ineffective unless conservative guardbands are applied, increasing system size and weight. We presented an automated technique based on symbolic simulation that determines a more aggressive peak power and energy requirement for a ULP processor for a given application. We show that the application-specific peak power and energy requirements determined by our technique are more accurate, and therefore less conservative, than those determined by conventional techniques. On average, the peak power requirements determined by our technique are 27%, 26%, and 15% lower than those generated based on design specifications, a stressmark, and profiling, respectively. Peak energy requirements generated by our technique are 47%, 26%, and 17% lower, on average, than those generated based on design specifications, a stressmark, and profiling, respectively. We also show that our technique can be used to guide optimizations that target and reduce the peak power of a processor. Optimizations suggested by our technique reduce peak power by up to 10% for a set of benchmarks.

ACKNOWLEDGMENTS

This work was supported in part by NSF, SRC, and CFAR, within STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. The authors would like to thank anonymous reviewers and Professor Lizy John for their suggestions and feedback. The authors would also like to thank Himanshu Shekhar Sahoo, who performed peak power and energy testing of ULP processors for Section 2.

REFERENCES

Allaboutbatteries.com. 2015. Battery Energy. Retrieved November 4, 2017 from http://www.allaboutbatteries.com/Battery-Energy.html. (2015).

ARM Mbed. Welcome to Mbed. Retrieved November 6, 2017 from https://www.mbed.com/en/.

John Greenough. 2015. The Internet of Everything: 2015 [Slide Deck]. Business Insider. http://www.businessinsider.com/internet-of-everything-2015-bi-2014-12

Jacob Borgeson. 2012. Ultra-low-power pioneers: TI slashes total MCU power by 50 percent with new "Wolverine" MCU platform. *Texas Instruments White Paper.* Retrieved November 6, 2017 from http://www.ti.com/lit/wp/slay019a/slay019a.pdf.

Randal E. Bryant. 1991. Symbolic simulation – Techniques and applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference.* ACM, 517–521.

Isidor Buchmann. 2016. The secrets of battery runtime. Battery University. Retrieved August 15, 2016 from http://batteryuniversity.com/learn/archive/the_secrets_of_battery_runtime.

Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Communications of the ACM* 56, 2, 82–90. DOI: http://dx.doi.org/10.1145/2408776.2408795

Cadence. 2014. *Encounter Digital Implementation User Guide.* Version: 14.11. http://www.cadence.com/.

B. H. Calhoun, S. Khanna, Yanqing Zhang, J. Ryan, and B. Otis. 2010. System design principles combining sub-threshold circuit and architectures with energy scavenging mechanisms. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*. 269–272. DOI: http://dx.doi.org/10.1109/ISCAS.2010.5537887

Hari Cherupalli, Rakesh Kumar, and John Sartori. 2016. Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems. In *43th Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE.

Cloud Compiling. 2013. Welcome to Cloud Compiling. Retrieved November 6, 2017 from http://www.cloudcompiling.com/.

Adam Dunkels, Joakim Eriksson, Niclas Finne, Fredrik Osterlind, Nicolas Tsiftes, Julien Abeillé, and Mathilde Durvy. 2012. Low-power IPv6 for the Internet of Things. In *9th International Conference on Networked Sensing Systems (INSS'12)*. IEEE, 1–6.

EEMBC. 2017. Embedded Microprocessor Benchmark Consortium. Retrieved November 6, 2017 from http://www.eembc.org.

Dave Evans. 2011. The Internet of Things: How the next evolution of the Internet is changing everything. Retrieved August 15, 2016 from https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

Tao Feng, L. C. Wang, Kwang-Ting Cheng, M. Pandey, and M. S. Abadir. 2003. Enhanced symbolic simulation for efficient verification of embedded array systems. In *Proceedings of the 2003 ASP-DAC Asia and South Pacific Design Automation Conference*. 302–307. DOI: http://dx.doi.org/10.1109/ASPDAC.2003.1195032

Kjartan Furset and Peter Hoffman. 2011. High pulse drain impact on CR2032 coin cell battery capacity. *Nordic Semiconductor and Energizer*. Retrieved August 15, 2016 from https://m.eet.com/media/1121454/c0924post.pdf.

O. Girard. 2013. OpenMSP430 project. *Opencores.org*. Retrieved March 1, 2014 from https://opencores.org/project,openmsp430.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 2001 IEEE International Workshop Workload Characterization (WWC'01)*. 3–14.

K. Hamaguchi. 2001. Symbolic simulation heuristics for high-level design descriptions with uninterpreted functions. In *Proceedings of the 6th IEEE International High-Level Design Validation and Test Workshop*. 25–30.

Michael S. Hsiao. 1999. Peak power estimation using genetic spot optimization for large VLSI circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM, 38.

Michael S. Hsiao, Elizabeth M. Rudnick, and Janak H. Patel. 1997. K2: An estimator for peak sustainable power of VLSI circuits. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*. IEEE, 178–183.

ITRS. 2015. International Technology Roadmap for Semiconductors 2.0 2015 Edition Executive Report. 2015. Retrieved November 6, 2017 from http://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_itrs/.

P. Jain and G. Gopalakrishnan. 1994. Efficient symbolic simulation-based verification using the parametric form of Boolean expressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 8, 1005–1015. DOI: http://dx.doi.org/10.1109/43.298036

Ramkumar Jayaseelan, Tulika Mitra, and Xianfeng Li. 2006. Estimating the worst-case energy consumption of embedded software. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE, 81–90.

Youngtaek Kim, Lizy Kurian John, Sanjay Pant, Srilatha Manne, Michael Schulte, W. Lloyd Bircher, and Madhu S. Sibi Govindan. 2012. AUDIT: Stress testing the automatic way. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, Washington, DC, USA, 212–223. DOI: http://dx.doi.org/10.1109/MICRO.2012.28

A. Kolbi, J. Kukula, and R. Damiano. 2001. Symbolic RTL simulation. In *Proceedings of the 2001 Design Automation Conference*. 47–52. DOI: http://dx.doi.org/10.1109/DAC.2001.156106

Vasileios Kontorinis, Amirali Shayan, Dean M. Tullsen, and Rakesh Kumar. 2009. Reducing peak power with a table-driven adaptive processor core. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, USA, 189–200. DOI: http://dx.doi.org/10.1145/1669112.1669137

L. Liu and S. Vasudevan. 2011. Efficient validation input generation in RTL by hybridized source code analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE'11)*. 1–6. DOI: http://dx.doi.org/10.1109/DATE.2011.5763253

Michele Magno, Luca Benini, Christian Spagnol, and E. Popovici. 2013. Wearable low power dry surface wireless sensor node for healthcare monitoring application. In *IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob'13)*. IEEE, 189–195.

Jeremy Morse, Steve Kerrison, and Kerstin Eder. 2016. On the infeasibility of analysing worst-case dynamic energy. *arXiv Preprint arXiv:1603.02580*.

K. Najeeb, V. Vardhan, R. Konda, S. Kumar, S. Hari, V. Kamakoti, and V. M. Vedula. 2007. Power virus generation using behavioral models of circuits. In *25th IEEE VLSI Test Symposium*. 35–42. DOI: http://dx.doi.org/10.1109/VTS.2007.49

National Instruments. 2016. Compile Faster with the LabVIEW FPGA Compile Cloud Service. Retrieved November 6, 2017 from http://www.ni.com/white-paper/52328/en/.

Intel Corporation. 2000. Intel Pentium 4 Processor in the 423-pin Package Thermal Design Guidelines. Retrieval August 15, 2016 from http://download.intel.com/support/processors/pentium4/sb/24920301.pdf.

IC Insights. 2005. Microcontroller Sales Regain Momentum After Slump. Retrieval date August 15, 2016 from http://www.icinsights.com/news/bulletins/Microcontroller-Sales-Regain-Momentum-After-Slump/.

J. A. Paradiso and T. Starner. 2005. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing* 4, 1, 18–27. DOI: http://dx.doi.org/10.1109/MPRV.2005.9

Chulsung Park, Pai H. Chou, Ying Bai, Robert Matthews, and Andrew Hibbs. 2006. An ultra-wearable, wireless, low power ECG monitoring system. In *IEEE Biomedical Circuits and Systems Conference (BioCAS'06)*. IEEE, 241–244.

Gil Press. 2014. Internet of Things by the numbers: Market estimates and forecasts. *Forbes*. Retrieval Aug 15, 2016 from https://www.forbes.com/sites/gilpress/2014/08/22/internet-of-things-by-the-numbers-market-estimates-and-forecasts/.

Sriram Sambamurthy, Sankar Gurumurthy, Ramtilak Vemu, and Jacob A. Abraham. 2009. Functionally valid gate-level peak power estimation for processors. In *Quality of Electronic Design (ISQED'09)*. IEEE, 753–758.

J. Sartori and R. Kumar. 2009. Distributed peak power management for many-core architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE'09)*.1556–1559. DOI: http://dx.doi.org/10.1109/DATE.2009.5090910

Kiran Seth, Aravindh Anantaraman, Frank Mueller, and Eric Rotenberg. 2006. Fast: Frequency-aware static timing analysis. *ACM Transactions on Embedded Computing Systems* 5, 1, 200–224.

Synopsys. 2015. *Design Compiler User Guide*. Version: K-2015.06. http://www.synopsys.com/.

Synopsys. 2015. *PrimeTime User Guide*. Version: K-2015.06-SP2. http://www.synopsys.com/.

Russell Tessier, David Jasinski, Atul Maheshwari, Aiyappan Natarajan, Weifeng Xu, and Wayne Burleson. 2005. An energy-aware active smart card. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13, 10, 1190–1199.

Texas Instruments. 2013. eZ430-RF2500-SEH solar energy harvesting development tool user's guide. Retrieved November 6, 2017 from http://www.ti.com/lit/ug/slau273d/slau273d.pdf, (2013).

Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. 2015. Worst-case energy consumption analysis for energy-constrained embedded systems. In *27th Euromicro Conference on Real-Time Systems*. IEEE, 105–114.

Chuan-Yu Wang and Kaushik Roy. 1998. Maximum power estimation for CMOS circuits using deterministic and statistical approaches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6, 1, 134–140.

Wikipedia. 2016. List of wireless sensor nodes. Retrieved November 6, 2017 from https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes.

Ross Yu and Thomas Watteyne. 2013. Reliable, low power wireless sensor networks for the Internet of Things: Making wireless sensors as accessible as web servers. *Linear Technology* Retrieved November 6, 2017 from http://cds.linear.com/docs/en/white-paper/wp003.pdf.

Bo Zhai, Sanjay Pant, Leyla Nazhandali, Scott Hanson, Javin Olson, Anna Reeves, Michael Minuth, Ryan Helfand, Todd Austin, Dennis Sylvester, and others. 2009. Energy-efficient subthreshold processor design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 8, 1127–1137.

Y. Zhang, Z. Chen, and J. Wang. 2012. Speculative symbolic execution. In *IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE'12)*. 101–110. http://dx.doi.org/10.1109/ISSRE.2012.8

Received September 2017; accepted September 2017