Enabling Effective Module-oblivious Power Gating for Embedded Processors

Hari Cherupalli†, Henry Duwe‡, Weidong Ye ‡, Rakesh Kumar‡, and John Sartori†
†University of Minnesota, ‡University of Illinois at Urbana-Champaign

***Abstract*—The increasingly-stringent power and energy requirements of emerging embedded applications have led to a strong recent interest in aggressive power gating techniques. Conventional techniques for aggressive power gating perform module-based power gating in processors, where power domains correspond to RTL modules. We observe that there can be significant power benefits from module-oblivious power gating, where power domains can include an arbitrary set of gates, possibly from multiple RTL modules. However, since it is not possible to infer the activity of module-oblivious power domains from software alone, conventional software-based power management techniques cannot be applied for module-oblivious power gating in processors. Also, since module-oblivious domains are not encapsulated with a well-defined port list and functionality like RTL modules, hardware-based management of module-oblivious domains is prohibitively expensive. In this paper, we present a technique for low-cost management of module-oblivious power domains in embedded processors. The technique involves symbolic simulation-based co-analysis of a processor's hardware design and a software binary to derive profitable and safe power gating decisions for a given set of module-oblivious domains when the software binary is run on the processor. Our technique is automated, does not require programmer intervention, and incurs low management overhead. We demonstrate that module-oblivious power gating based on our technique reduces leakage energy by 2× with respect to state-of-the-art aggressive module-based power gating for a common embedded processor.**

I. Introduction

A large number of existing and emerging computing applications require ultra-low-power operation and extreme energy efficiency [1], [2], [3], [4], [5], [6]. Notable among these are the internet of things, sensor networks, wearables, and health monitors. The 2015 ITRS report projects power and energy constraints of these systems to be even tighter in the future [7]. Unsurprisingly, these applications rely on low-power microcontrollers and microprocessors that have become the most widely-used type of processor in production today [8], [9], [10].

The ultra-low power and energy requirements of emerging applications, along with the increasing leakage energy dissipation that has accompanied CMOS scaling [11], have fueled interest in aggressive power gating techniques. Conventional aggressive power gating techniques perform module-based power gating, i.e., power gating of RTL modules during periods of inactivity [12], [13], [14]. An RTL module is encapsulated with a well-defined port list, making it relatively easy to determine when a module is inactive based on input signals in the port list.

While RTL modules form convenient boundaries for defining power domains, module-based domains may not be the best option for supporting aggressive power gating. Logic is grouped into a module based on common functionality, not necessarily based on correlated activity. In several cases, activity of logic in the same module can have uncorrelated activity (e.g., different registers in the register file may not be used by the same instruction or even the same application), while logic in different modules can often be correlated (e.g., when one module feeds data or control signals to another).

In this paper, we make a case for aggressive power gating based on *module-oblivious* power domains. A module-oblivious power domain is an arbitrary set of gates that have correlated activity. Module-oblivious power domains may contain only a subset of gates in a module, may contain gates from multiple modules, and may also consist of logic from non-microarchitectural modules (e.g., uncore, debug logic, peripherals, etc.). The goal of grouping logic into module-oblivious power domains based on correlated activity rather than module membership is to enable **larger segments of logic** to be power gated for **longer periods of time**, thus **saving more energy**.

While module-oblivious power domains may provide more opportunities to reduce power, conventional hardware and software-based power management techniques cannot manage these unconventional domains. A hardware or software-based power gating management technique must be able to guarantee that a domain is idle before it is powered off and that an idle domain is powered on before it will be used. Since the activity of an arbitrary collection of gates that may constitute portions of multiple modules cannot be inferred based on software alone, module-oblivious domains cannot be managed in software using conventional techniques. Hardware-based power management detects when a domain is idle, then powers off the domain. Since a module-oblivious domain is not encapsulated with a well-defined port list and does not have a well-defined function but instead consists of an arbitrary collection of gates that can contribute to many different functionalities, detecting when the domain is idle requires monitoring of all input nets to the gates in the domain. The high overhead of monitoring the activity of so many signals easily outweighs the benefits of power gating. Any viable technique for managing module-oblivious power domains must be able to infer the gate-level activity induced by software, so that the prohibitive overheads associated with hardware monitoring of an arbitrary set of gates can be avoided.

In this paper, we propose a technique that generates safe, aggressive power gating management decisions for module-oblivious power domains. The gate-level activity profile of an application is captured through a symbolic simulation of the application's binary that characterizes domain activity for all possible application inputs. Safe power gating decisions are then generated such that each domain is guaranteed to be powered on by the time it is used, and domains are aggressively powered off whenever profitable. Power gating decisions are then embedded into the application binary. This software-based power management approach avoids the prohibitive overheads of managing module-oblivious domains in hardware.

Our proposed technique is automated, requires no programmer intervention, and incurs low management overhead. Also, while the technique is general, it is best suited for embedded systems. Embedded system designers routinely perform hardware/software co-design [15], [16] or license hardware IP [17], [18], so they often have access to both RTL and software binary – the inputs needed by our power gating framework.[1] Also, embedded processors and applications tend to be simple, so our symbolic simulation-based analysis scales well in such settings.

This paper makes the following contributions.

- We make a case for module-oblivious power gating. We show that module-oblivious power gating can result in 2× higher leakage energy savings compared to state-of-the-art module-based power gating.

[1]Power gating binary annotation can be offered as a cloud compilation service by the hardware system vendor in non-embedded settings, where the application developer does not have access to the processor description [19], [20], [21].

• To enable module-oblivious power gating, we present a fully-automated technique that performs co-analysis of an embedded system's processor netlist and application binary to make safe, aggressive power gating decisions.[2] To the best of our knowledge, this is the first technique for module-oblivious power gating.

• We fully implement module-oblivious and module-based power gating in openMSP430 using an industry-standard UPF methodology that accounts for all power gating overheads. We demonstrate that module-oblivious power gating can achieve 2× higher leakage energy savings compared to module-based power gating. We show that module-oblivious gating based on our techniques achieves leakage energy savings that are within 8% of optimal.

• Finally, we show that our technique for managing module-oblivious domains is effective even at managing conventional module-based domains. It saves 12% more energy than an idealized implementation of IdleCount – a hardware-based domain management technique for module-based domains. Our benefits are within 6% of optimal for module-based domains.

II. Related Work

While a large body of work exists on processor- and core-level power gating [12], [13], [28], [29], [30], [31], emerging power- and energy-constrained applications have fueled recent work on aggressive module-based power gating techniques [14], [30], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42]. These techniques, including those that adaptively re-size microarchitecural structures [14], [13], and techniques that target uncore components (e.g., on-chip routers [43], [44], [45]), focus on power gating of RTL modules. Since module-based domains are smaller and more homogeneous, they may provide more frequent opportunities for aggressive power gating than processor- or core-level solutions. Table I shows the number of power domains supported in some recent microprocessors / microcontrollers. As can be seen, power gating is already being performed aggressively; many processors have a large number of power domains.

TABLE I: Power Domains in Recent Processors

Processor	#domains
TI MSP430 Wolverine	"many" [22]
ARM Cortex-A9	14 [23]
ARM Cortex-A15	8 [24]
Atmel SAML21	5 [25]
Intel Atom E6	15+ [26], [27]

In this paper, we make a case that aggressive power gating may save even more leakage energy when power domains are module-oblivious rather than module-based. Primarily, this is because logic in an RTL module is grouped together based on common functionality, not necessarily a correlated activity profile, whereas logic in a module-oblivious domain is grouped together based on common periods of inactivity that allow power gating to be performed. To best of our knowledge, this is the first work on module-oblivious power gating.

In terms of power domain management, prior works have used software- or hardware-based management. Software-based domain management techniques [46], [14] infer when power domains will be inactive by analyzing an application binary. This requires the functionality of managed power domains to be software-visible. Prior techniques for software-based power domain management cannot be used for module-oblivious power domains, because such domains may contain logic that belongs to many modules and contributes to many fine-grained functionalities, making it impossible to infer activity of module-oblivious domains from software alone.

[2]Our automated co-analysis tool for module-oblivious power gating is available for download at the following link: http://people.ece.umn.edu/users/jsartori/tools.html

Hardware-based domain management techniques use hardware monitors to detect when power domains are inactive [28], [13], [29], [42], [36], [37]. Such an approach is feasible for prior works on aggressive module-based power gating, because an RTL module is encapsulated with a well-defined interface (port list) and function. Thus, it is possible to infer domain activity from a relatively small number of signals. Hardware monitoring is infeasible for module-oblivious domains, however, because they do not have a well-defined interface or functionality. As such, the number of signals that must be monitored to infer domain activity is prohibitively large.

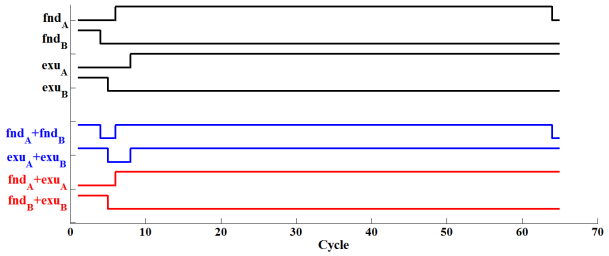
Symbolic simulation has been applied in circuits for logic and timing verification, sequential test generation [47], [48], [49], [50], [51], and determination of application-specific V_{min} [52]. It has also been applied for software verification [53]. However, to the best of our knowledge, no existing technique has applied symbolic simulation for power gating and power domain management.

III. Motivation

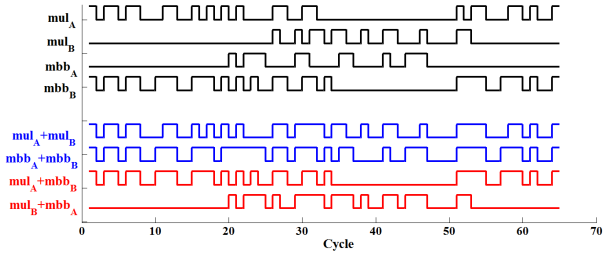
A. A Case for Module-oblivious Power Domains for Microprocessors

There are several reasons why module-oblivious power domains may provide significantly more opportunities for power gating than module-based domains in microprocessors. One reason is that logic in microarchitectural modules is grouped together largely based on functionality or position in the processor pipeline, which does not necessarily imply correlation in terms of activity. It may often be the case that different logic partitions within the same microarchitectural module have very different activity profiles. For example, many microarchitectural modules support "one-hot" logic. This implies that each logical state is mutually exclusive of all other states. Similarly, each instruction selects and executes on one execution unit. This leaves all other execution units idle. Furthermore, it is common for several modules to have parts that are nearly always active and other parts that are nearly always idle. This weak or anti-correlation between the activity profiles of different parts within a module limits the effectiveness of power gating for module-based domains. Figure 1a shows activity profiles for the frontend and execution unit modules of an openMSP430 processor [54] running an encryption application (tea8), where each module has been divided into two sub-modules. In the figure, a high/low value indicates that a sub-module is active/idle. Since both the frontend and the execution unit have at least one part active during nearly every instant of this time period, there is no opportunity to power gate either module. Stated differently, fnd_A and exu_A prevent the frontend and execution unit from being power gated, even though fnd_B and exu_B are almost completely inactive. If, however, fnd_A and exu_A were combined to form one power domain and fnd_B and exu_B formed a second domain, the second domain could be power gated during this time period. Uncorrelated activity within modules and correlated activity across modules indicates that there may be significant opportunities to perform more aggressive power gating with module-oblivious power domains.

Another reason for correlated activity across module boundaries is that logic in one module often drives logic in another module. Although the entire modules are unlikely to have correlated activity, the driving and driven parts of the modules do have highly correlated activity. Also, such logical components are typically in close proximity in a chip layout, making them good candidates to be placed in the same domain for power gating. Figure 1b illustrates this behavior with an example for an application that performs multiplication, where the multiplier and memory backbone modules of the openMSP430 processor have each been divided into two sub-modules. Sub-module mbb_A contains the peripheral data input bus that feeds input data to the multiplier (since the multiplier is one of the peripherals). The



(a) Uncorrelated activity within a module can prevent power gating of module-based domains, whereas module-oblivious domains allow more aggressive power gating.



(b) When one module drives another, the driving and driven logic belong to different modules but have highly correlated activity, whereas logic within the same module may have completely uncorrelated activity.

Fig. 1: Activity profiles for different module partitions suggests that module-oblivious domains may provide significantly more opportunities for power gating than module-based domains.

activity of this sub-module is highly correlated to that of mul_B, which contains the input side of the multiplier. When domain wakeup overhead is considered, module-based power domains do not allow any power gating of these modules (blue activity profiles). However, when module-oblivious domains are formed (red activity profiles), both module-oblivious domains can be power gated for significant portions of this time period.

Figure 2 is a correlation matrix that shows the correlation between each pair of sub-modules in the openMSP430 processor, where each module has been partitioned into four sub-modules and correlation equals the fraction of cycles in which two sub-modules exhibit the same activity (active or idle). The dashed boxes along the main diagonal encircle correlation scores for sub-modules that belong to the same module. It can be observed that not all parts of a module have correlated activity, and in many cases, different parts of the same module have highly uncorrelated activity. Tracing down a row corresponding to a given sub-module, it can be observed that there *always* exist one or more sub-modules from different modules that have more correlated activity profiles than a sub-module from the same module. For example, in the row showing correlations for the last sub-module in the frontend, we have encircled all the (10) sub-modules from different modules that are more strongly correlated to this frontend sub-module than *any* of the other frontend sub-modules. These observations suggest that module-based power domains may often miss opportunities to power gate idle logic, whereas module-oblivious power domains may provide significantly more opportunities to power gate larger areas of logic for longer periods of time. In Section VI, we show that a full-fledged UPF implementation of openMSP430 with module-oblivious domains achieves up to 2× more leakage savings than an implementation with module-based domains.

B. A Case for a Novel Management Technique for Module-oblivious Domains

Reaping the power benefits enabled by module-oblivious domains requires a power domain management technique that can determine

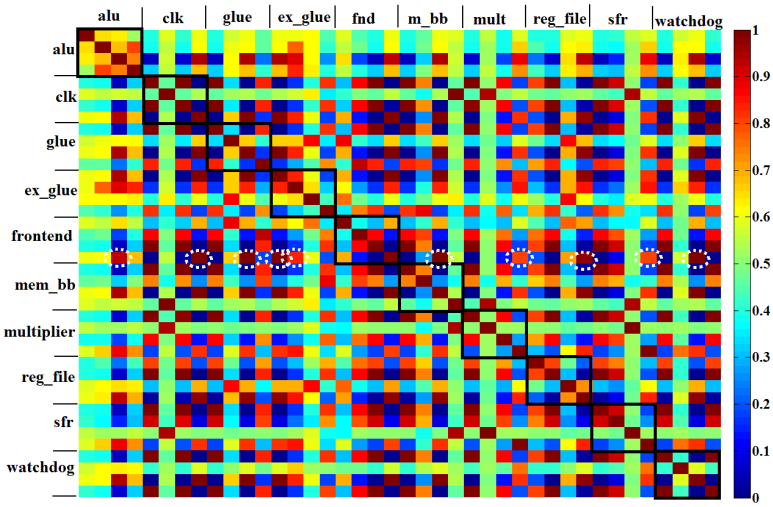


Fig. 2: Different parts of the same module can have uncorrelated activity profiles. For nearly all sub-modules, a sub-module from a different module has more correlated activity than a sub-module from the same module.

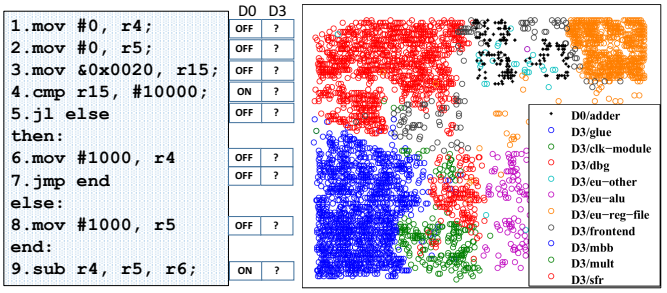


Fig. 3: It is possible to infer the activity of a module-based domain (e.g., D0 – the adder in the execution unit) based on software alone. It is not possible to infer the activity of a module oblivious domain (e.g., D3 from Figure 4c) based on software alone.

when domains are idle / active and power them off / on accordingly. Unfortunately, existing techniques that manage module-based domains through software or hardware cannot be used for module-oblivious domains. Consider existing software-based management techniques. Software-based management is possible when domain activity can be inferred from software, as is the case for many module-based domains [12]. In the example code listing in Figure 3, domain D0 is a module-based domain corresponding to the adder in the execution unit. Since the adder module has a well-defined architectural function, it is possible to infer when the domain must be powered on. For example, instructions 4 (compare) and 9 (subtraction) use the adder, so domain D0 must be powered on when those instructions reach the execution stage. The adder can potentially be powered off for other instructions, since they do not use the adder.

For a module-oblivious domain, however, it is not possible to infer domain activity from software alone. A module-oblivious domain does not have a well-defined architectural function. It is a collection of gates with correlated activity profiles that may belong to many modules and contribute to many functionalities. For example, domain D3 in Figure 3 corresponds to the module-oblivious domain in Figure 4c (see Section V-A for details of how module-oblivious domains are constructed). The domain contains gates from ten different modules, including glue logic, the memory backbone, and clock generation logic for which activity cannot be inferred based on software.

Similarly, existing hardware-based domain management techniques

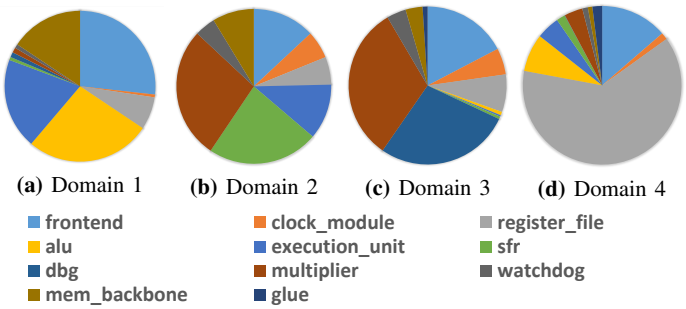


Fig. 4: Breakdown of domain composition for each module-oblivious power domain. All four domains have gates from at least eight microarchitectural modules.

```
module domain_activity_detector_D0 (
 inst_type, // from decode
 wkup_adder);
input [11:0] inst_type;
output wkup_adder;
wire wkup_adder = {inst_type[`ADD] |
 inst_type[`SUB] | inst_type[`ADDC] |
 inst_type[`SUBC] | inst_type[`CMP] |
 inst_type[`REL_JMP] | inst_type[`RETI]};
endmodule
```

(a) Verilog statements for inferring the activity of the execution unit adder module – synthesizes to 6 gates.

```
module domain_activity_detector_D3 (
 in,   // domain inputs
 ff_d,  // domain ff D-pins
 ff_q,  // domain ff Q-pins
 clk, wake_up_domain );
input [704:0] in; input [648:0] ff_d;
input [648:0] ff_q; input clk;
output wake_up_domain; reg [704:0] in_delay;

always @ (posedge clk)
begin
        in_delay <= in;
end

wire [704:0] in_toggled = in ^ in_delay;
wire [648:0] ff_toggled = ff_d ^ ff_q;
wire any_input_toggled = {|in_toggled};
wire any_ff_toggled = {|ff_toggled};
wire wake_up_domain = {any_input_toggled | any_ff_toggled};
endmodule
```

(b) Verilog statements for inferring the activity of the module-oblivious domain from Figure 4c – synthesizes to 4010 gates.

Fig. 5: Hardware-based domain management logic for a module-based domain can be relatively simple, whereas domain management logic for a module-oblivious domain is prohibitively expensive.

are infeasible for module-oblivious domains. Hardware-based domain management dynamically determines when a power domain is idle / active based on processor control signals. This can be relatively straightforward for module-based designs, since RTL modules are encapsulated, with a well-defined interface (port list) and functional description. For example, consider D0 in Figure 3 – the adder module. To determine if this domain will be active, hardware-based management logic only needs to detect if a decoded opcode corresponds to one of the instructions that uses the adder. Figure 5a shows the verilog statements that can be added to the decode stage to infer the activity of the adder. Synthesized, this logic corresponds to only 6 gates.

On the other hand, domain management logic for a module-oblivious domain is not simple. Since module-oblivious domains are not nicely encapsulated with a well-defined interface and function, the only way to infer their activity in hardware is to monitor activity on all input nets that cross the domain boundary. Additionally, state elements (flip-flops) inside the domain must be monitored for activity, since a state machine inside the domain could be active even without triggering any activity at the domain boundary. Figure 5b shows

TABLE II: Overheads for hardware-based management of module-oblivious power domains in openMSP430.

Domains	Gate Count	FF Count	Domain Inputs	Area Overhead	Static Power Overhead
2	6695	784	947	143.36%	136.44%
3			1203	159.38%	151.30%
4			1450	183.52%	173.69%

the verilog statements that infer the activity of the module-oblivious domain D3 from Figure 4c. Synthesized, this logic corresponds to 4010 gates.

The overhead of managing module-oblivious domains in hardware becomes prohibitive when the full processor is considered. Table II shows the area overhead incurred by hardware-based domain management logic in openMSP430 for two-, three-, and four-domain designs. The overheads preclude any possible benefits from aggressive power gating, prohibiting the use of hardware-based domain management for module-oblivious power gating.

Any viable technique for managing module-oblivious power domains must be based on inferring their gate-level activity from software, such that the prohibitive overheads associated with hardware-based monitoring of an arbitrary set of gates can be avoided. In the next section, we describe a low-overhead technique based on hardware-software co-analysis that can infer the activity of module-oblivious domains to enable aggressive module-oblivious power gating.

IV. A Co-Analysis Based Approach for Module-Oblivious Power Gating

A power domain management technique must infer domain activity to determine when domains can be powered off, while guaranteeing that that they will be powered on when active. Since a module-oblivious domain may contain an arbitrary set of gates, inferring domain activity requires gate-level analysis of software execution on a processor. Activity analysis cannot be based on profiling (i.e., observing activity for several benchmark runs with different input sets), since profiling is input-specific and may result in incorrect management decisions when in-field inputs are different than the inputs characterized during profiling. An incorrect management decision is unacceptable, since it may lead to incorrect program execution (e.g., when a domain needed by the program is turned off). Below, we describe a novel technique that uses symbolic simulation to characterize the gate-level activity of an application on a processor to generate power gating decisions for module-oblivious power domains. The symbolic simulation uses unknown logic values (Xs) for all inputs so that the generated activity profile characterizes all possible executions of the application for all possible inputs. We use the results of input-independent activity analysis to generate instruction-level power domain management decisions that achieve near-optimal power benefits while *guaranteeing* that all domains are powered on whenever needed. Figure 6 provides an overview of our module-oblivious power gating technique.

A. Gate Activity Analysis

The first stage of our module-oblivious domain management technique infers the activity of power domains during an application's execution. Normally, a gate-level simulation could infer the activity of all processor gates for only one input set. However, our technique propagates Xs for all application inputs, allowing us to infer the activity of all gates for *all possible input sets*. Combined with the domain mapping that specifies which gates belong to each domain, we can infer domain activity for all possible executions of an application on a processor.

Gate Activity Analysis (Algorithm 1) performs symbolic simulation [47] of an application binary running on the gate-level netlist of the processor, in which unknown logic values (Xs) are propagated for

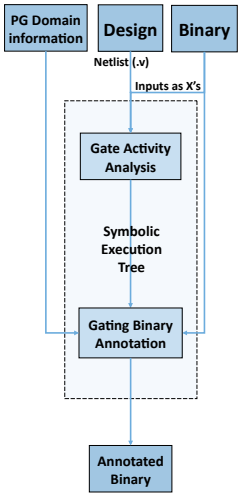


Fig. 6: Our analysis generates input-independent power gating decisions for module-oblivious power domains.

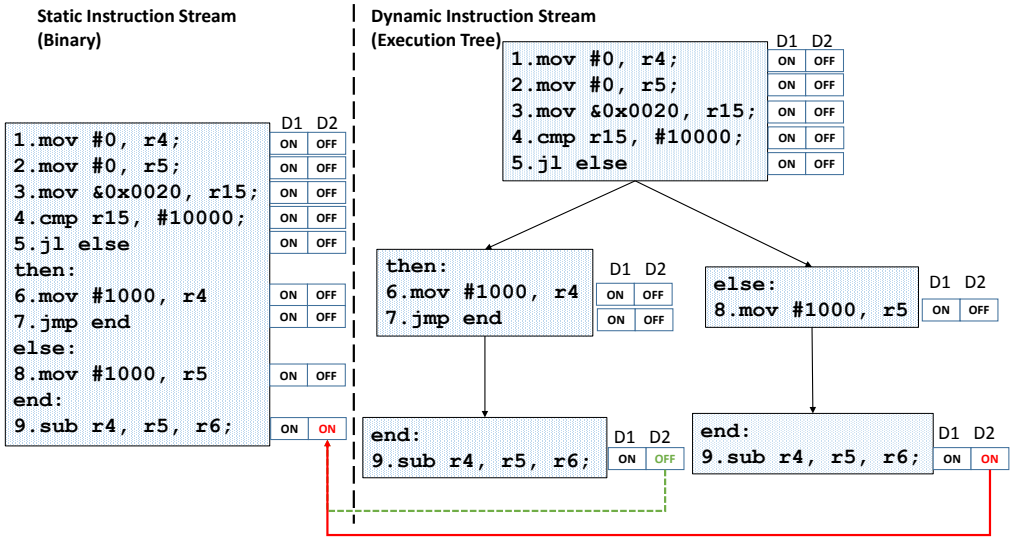


Fig. 7: Illustration of Gating Binary Annotation for an example code (an if-else block). For simplicity, we show each instruction as taking a single cycle, only show domain-level activity, and use a wake-up latency of zero cycles.

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. 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 re-simulated 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 un-simulated state is loaded from the last 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), Gate Activity Analysis is complete.[3]

During symbolic simulation, the simulator captures the activity of each gate at each point in the execution tree. A gate is considered active in a particular cycle 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 (and by extension, all instances in which each domain could possibly be active) for all possible executions of the application. As such, it also describes when power domains (even module-oblivious domains) can be safely powered down and when they must be powered up. The next section describes how inferred domain activity information is translated into domain management decisions.

[3]For our benchmarks (Table III), analysis takes 37 minutes, on average, and a maximum of 2 hours for our largest benchmark. While naive symbolic gate-level simulation does not scale well to large, complex applications or processors, an increasing number of future applications for the internet of things, wearables, embedded sensors, and other ultra-low-power domains are expected to continue to use simple applications and processors [7], [10]. Also, several heuristics have been shown to be effective in enabling symbolic analysis for complex applications and processors [55], [56].

Algorithm 1 Gate Activity Analysis

```
1.  Procedure Generate 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 un-processed 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) // perform simulation for this cycle
13.         e.annotate_gate_activity(e,e') // annotate tree point with activity
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.         foreach a ∈ possible_PC_next_vals(e) do
19.             if a ∉ T then
20.                 e' ← e.update_PC_next(a)
21.                 U.push(e')
22.                 T.insert(a)
23.             end if
24.         end for
25.     end if
26. end while
```

B. Gating Binary Annotation

Gating Binary Annotation (GBA) takes as input the annotated symbolic execution tree from Gate Activity Analysis, gate-to-domain mapping information, and domain wake-up overheads, and produces a binary in which each static instruction is annotated with power gating decisions for all domains in the processor. Algorithm 2 describes GBA. GBA considers each path through the symbolic execution tree.[4] During each cycle of a path's execution, GBA determines which domains can have active gates and thus must be powered on. To ensure safety, GBA also marks a domain as active during the N cycles leading up to a period of activity, where N is the wakeup latency required to power up the domain. These cycle-level power gating decisions are mapped to all the static instructions that have dynamic instances in the pipeline during the wakeup cycles or the current cycle.

[4]For our benchmarks, GBA takes 11.71 seconds, on average, and a maximum of 35.44 seconds for our largest benchmark.

Algorithm 2 Gating Binary Annotation for Power Gating Control

Procedure *Annotate Binary with PG Decisions(annotated_symbolic_execution_tree, domain_mapping, domain_wakeup_overhead)*

```
1. P_SET ← enumerate all paths in annotated_symbolic_execution_tree
2. Mark all domains as idle for all instructions/addresses in the binary
3. foreach path p ∈ P_SET do
4.     foreach cycle c ∈ p do
5.         foreach gate g ∈ Processor do
6.             if g is toggled then
7.                 D ← domain_mapping.get_domain(g)
8.                 wo ← domain_wakeup_overhead.get(D)
9.                 I ← get_instructions_being_executed(p, c, wo)
10.                foreach i ∈ I do
11.                    Mark domain D as active at instruction i in binary
12.                end for
13.            end if
14.        end for
15.    end for
16. end for
```

Once GBA has considered each execution path through an execution binary, each static instruction has an annotation specifying which domains must be powered on when the instruction is in the decode stage. This annotation guarantees safety, because each possible dynamic instance of a static instruction is considered by GBA. If a domain is marked as being powered on for any dynamic instance of a static instruction, the static instruction is annotated with an "ON" decision for the domain. This is conservative to ensure safety, but it works well for embedded applications, which tend to have simple control flow. If a domain is not active for any dynamic instance of a particular instruction (even considering wakeup overheads), the domain is powered off. The annotated binary containing domain management decisions can be used to manage power domains using one of the several techniques described in Section IV-D.

C. Illustrative Example

This section illustrates the procedures for managing module-oblivious domains, described in Sections IV-A and IV-B, with an example. Figure 7 revisits the example code from Figure 3 to demonstrate that our technique based on hardware-software co-analysis *can* infer the activity of module-oblivious domains, which was impossible to infer from software alone.

Figure 7 shows the annotated symbolic execution tree generated by Gate Activity Analysis (GAA). GAA simulates the application starting at instruction 1. When an input value is read in instruction 3, instead of storing the input bits, unknown logic values (Xs) are stored in `r15`. During instruction 5, an X propagates to the PC inputs, since the result of the comparison in instruction 4 is unknown (X). At this point, a branch is created, and the simulation state is stored in a stack for later analysis with the address of instruction 8 (`else:`) in the PC inputs. Simulation continues through the left (`then:`) control flow path to completion, starting with instruction 6. After finishing instruction 9, the stored simulation state is popped off the stack and the right control flow path is simulated to completion, starting with instruction 8.

During simulation, GAA annotates each dynamic instruction with domain activity for each domain (D1 and D2 in Figure 7). ON means that at least one gate in the domain *might* be active during that instruction; OFF means that all of the domain's gates are guaranteed to be inactive for that instruction. Next, Gating Binary Annotation (GBA) maps the domain states (ON/OFF states) from the symbolic execution tree to the static instructions in the application binary. Consider static instruction 1 (`mov #0, r4`). There is only one dynamic instance of the instruction in the symbolic execution tree, and for this instance, domain D1 is ON and D2 is OFF. Therefore, GBA annotates the corresponding static instruction with the information that D1 is ON and D2 is OFF.

Now consider static instruction 9 (`sub, r4, r5, r6`). There are two dynamic instances of the instruction in the symbolic execution tree. The activity of D1 is consistent across the two instances (D1 is ON for both); therefore, GBA annotates the static instruction with the information that D1 is ON. The activity of D2, however, is not consistent across the two dynamic instances of instruction 9; D2 is OFF in one and ON in the other. In this case, GBA conservatively resolves the conflict by marking D2 as ON in the static instruction annotation. This ensures safety for all possible application executions.

D. Microarchitecture Support for Software-based Power Gating

Sections IV-A and IV-B describe a technique that can infer the activity of module-oblivious domains without costly hardware-based monitoring and use inferred domain activity to make safe and profitable domain management decisions. This section describes microarchitectural support for communicating domain management decisions to the control logic that powers the domains off and on.

Power Gating Instructions:
A straightforward way to generate power gating control signals is to insert instructions in the binary that direct power domains when to turn off and on. To ensure that a power domain is powered on before it is used, the wakeup instruction for a domain must arrive *wakeup-latency* cycles before an instruction, I_A, that will activate the domain. For an in-order processor, we insert the wakeup instruction *wakeup-latency* instructions ahead of I_A. This guarantees that the domain will be powered up even if instructions have variable latencies. A power down instruction for a domain is inserted immediately after the last instruction that specifies that the domain must be powered on. Since GBA marks domains as active (ON) during their entire wakeup and activity period, the wakeup instruction is simply inserted before the first instruction that marks a domain as ON, and the power down instruction is inserted after the last instruction that marks a domain as ON. For example, in Figure 7 an instruction that turns D1 ON and D2 OFF is inserted before instruction 1, while an instruction to turn D2 ON is inserted before instruction 9. Note that a similar support mechanism has been used in prior work on software-based power gating of functional units for embedded processors [46].

Reserved Instruction Bits:
Another option for indicating when domains should be powered on and off is to modify the ISA of the processor to reserve some bits in the instruction to indicate the ON/OFF state of each domain. The number of bits required is equal to the number of domains. The main benefit of this technique is that it does not require extra instructions to be inserted in the binary. However, since the number of bits that can be reserved in the instruction for power gating would likely be small, this technique can only support a small number of power domains. Also, reserving instruction bits for power-gating decisions may increase code size if the instruction length must be increased to accommodate the bits.

PC Monitoring:
Another alternative is to maintain a software-populated table that holds the addresses of annotated instructions, along with corresponding information about which domains should be turned ON or OFF when that instruction's address enters the PC. Every N instructions, the application populates the table with the addresses of annotated instructions in the next window of N instructions. When the PC matches one of the addresses in the table, the power domain control signals stored in that table entry are sent to the respective power domains to switch them on or off. This technique requires some software overhead to re-populate the table and hardware overhead to implement the table as a CAM.

E. Ensuring Correctness

The proposed approach guarantees correct execution of the application at three levels.

1. Guaranteeing that domains turn on when needed: Our co-analysis approach characterizes domain activity for all possible executions of an application for all possible inputs to the application. A power domain is only turned off if it is not used by an instruction in all execution paths through the code. (Section IV-B).

2. Guaranteeing that analysis is input-independent: We perform a symbolic simulation in which all application inputs are replaced by Xs. This ensures full characterization of application-induced activity on the processor for all possible application inputs (Section IV-A).

3. Guaranteeing that hardware implementation is correct: We use an automated industry-standard UPF flow to fully implement power gating designs and accurately account for all implementation overheads of power gating (Section V-C).

F. Generality

While we primarily target low-power processors used by numerous embedded applications [1], [2], [3], [4], [5], module-oblivious power gating can be applied in other contexts as well. We discuss generality below.

Complex Processors: More complex processors contain more performance-enhancing features such as large caches, prediction or speculation mechanisms, and out-of-order execution, that introduce non-determinism into the instruction stream. Co-analysis is capable of handling this added non-determinism 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 co-analysis already explores taken and not-taken paths for input-dependent branches, it can easily be adapted to handle branch prediction.

Although out-of-order execution appears to execute instructions in a non-deterministic order, the ordering of instructions is actually deterministic, based on the dependence pattern between instructions. While instructions may execute in different orders depending on the state of pipelines and schedulers, a processor that starts from a known reset state and executes the same piece of code will transition through the same sequence of states each time. Thus, modifying input-independent CFG exploration to perform input-independent exploration of the data flow graph (DFG) may allow analysis to be extended to out-of-order execution.

Multi-threading and Multi-programming: Multi-programming and multi-threading present challenges for application analysis, since they introduce non-determinism in the instruction stream executed by a processor. Since it may not be possible to determine all possible interleavings of instructions between threads, a minor adaptation to Algorithm 1 is needed to perform co-analysis for a thread that is agnostic to the behavior of other threads. Any state that is not maintained as part of a thread's context can be assumed to have a value of X when symbolic execution is performed for an instruction belonging to the thread. This approach generates safe power gating decisions for the thread irrespective of the behavior of the other threads.

Binary Compatibility: Software-based power gating techniques (not just ours) may have issues with binary compatibility due to inserted power gating instructions. We can address this by maintaining and distributing the un-instrumented binary and running a one-time co-analysis to tailor the binary for a specific processor.

System Code: Our evaluations have been performed in the context of bare-metal design (no OS). While many low-power microprocessors and a large segment of embedded systems are bare-metal systems (application running on the processor without an operating system (OS)) [57], [58], [59], [60], use of an OS is common in several embedded application domains, as well as in more complex systems. In such systems, system code must be analyzed in addition to application code to identify power gating opportunities. For relatively simply OSes (e.g., embedded and realtime OSes), it may be possible to completely analyze and annotate the OS using GAA+GBA. In some settings, it may not be possible to analyze system code completely. A simple solution to guarantee safety of power gating decisions in such settings is to save the domain state as part of the application context, turn on all domains before entering system mode, and return to the saved state when returning to user mode. The performance impact of wakeup overhead during a context switch should be small in such settings since OS invocations are relatively infrequent and wakeup latency is negligible compared to the time between context switches.

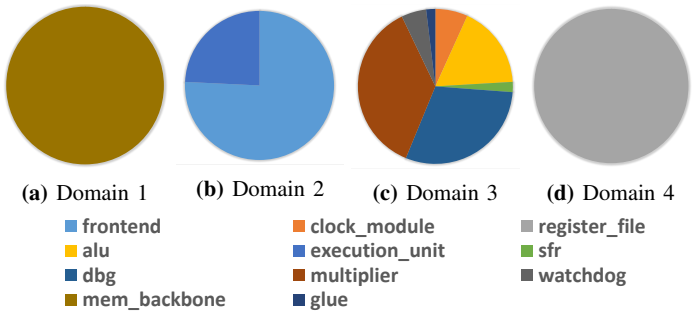


Fig. 8: Domain composition of module-based power domains. Each module belongs to only one domain.

V. Methodology

In this section, we first describe how we construct module-based domains and module-oblivious domains for our study. We then describe the different techniques we evaluate for managing power domains. Finally, we discuss other methodological details of our evaluations.

A. Constructing Power Domains

Module-based Domains: We construct module-based domains following the conventional approach for aggressive power gating, in which power domains are formed to encompass microarchitectural modules. When the number of modules is greater than the number of allowable power domains, modules are grouped together into domains using hierarchical agglomerative clustering [61]. This clustering technique combines a set of N clusters into N-1 clusters, based on an optimization objective. In this case, the objective function uses activity profiles for the clusters (obtained from benchmark profiling) to determine which combination of modules maximizes the potential energy savings achieved by power gating the resulting domains. Potential energy savings are measured in *gated cycles*, where one gated cycle corresponds to power gating one gate in the gate-level netlist for one cycle. Figure 8 shows the domain composition for four module-based domains that maximize leakage energy savings for module-based power gating.

Module-oblivious Domains: We use the same clustering technique as for module-based domains, with two key differences. First, whereas module-based domain construction begins with all processor *modules* in separate clusters and combines clusters using hierarchical agglomerative clustering to form the desired number of domains, module-oblivious domain construction begins with every *gate* in a separate cluster and combines clusters to form the desired number of domains. Since a gate may end up in a cluster containing gates from other modules, the resulting domains are module-oblivious.

Second, since an application's in-field inputs may not always match the inputs used during profiling, we use activity profiles produced by

input-independent gate activity analysis (Section IV-A) to identify correlated gates and generate power domains, instead of profiles captured assuming specific inputs. We treat an X in an activity profile as a toggle, since it indicates that a net could toggle for some possible input. Input-independent domain formation ensures robustness of domains across variations in an application's input set. We form domains using the activity profiles for only three randomly-selected applications in our benchmark set (tea8, binSearch, Autocorr), and use these domains to perform evaluations for all thirteen benchmarks in Table III. In practice, we envision that domains will be formed using activity profiles that are representative of a system's target workloads (similar to how benchmarks are used to determine microarchitectural parameters). The actual workloads that the processor will run in the field may be different and many more than the number of benchmarks used for domain formation. As such, we chose a small number of benchmarks for domain formation relative to the total number of applications used for evaluation. Nevertheless, our evaluations show significant benefits even for the ten benchmarks that were not used for domain formation (Section VI). This is because the correlated activity among gates in different modules is often ISA and microarchitectural implementation-dependent, so only a small number of benchmarks are needed to determine which gates have correlated activity profiles and form domains accordingly.

B. Power Domain Management

IdleCount [62] is a hardware-based power gating management technique that uses a counter per domain to count the number of cycles a domain has been idle. The counter is reset every cycle its domain is active. When the counter reaches a threshold, k, the domain is powered down. We perform evaluations for $k = 5$, 10, and 100. Although [62] only proposes power gating of functional units, we optimistically evaluate it for any arbitrary processor modules, even software-invisible modules. We only evaluate IdleCount for module-based domains, since activity monitoring is prohibitively expensive for module-oblivious domains (Section III-B). To make this baseline even more optimistic, we do not consider the overhead of implementing hardware-based domain monitoring logic for software-invisible module-based domains.

Oracular Management assumes perfect knowledge of an application's inputs to determine exactly when every power domain should be powered on or off to maximize energy savings. Each domain is woken up just in time so that the domain is fully powered on by the first cycle that any of its gates become active (i.e., toggle). Oracular management powers down a domain immediately whenever profitable, i.e., when all gates in the domain will be idle for at least the number of cycles it takes the domain to wake up. The benefits of oracular management represent the upper bound on the benefits that can be achieved by any power gating technique.

Our approach: Software-Hardware Co-analysis uses input-independent symbolic simulation to annotate instructions in the application binary with power gating decisions for each domain, as described in Section IV.

C. Simulation Infrastructure and Benchmarks

We verify our approach on a silicon-proven general purpose processor – openMSP430 [54]. [5] Since MSP430 supports aggressive module-based power gating [22], it is a suitable testbed for comparison of module-based and module-oblivious power gating. The processor is synthesized, placed and routed in TSMC 65GP (65nm) technology at an operating point of 1V and 100 MHz using Synopsys Design Compiler [64] and Cadence EDI System [65]. Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using a custom gate-level simulator to efficiently traverse the control flow graph of an application and capture input-independent activity profiles (Section IV-A). We show results for all benchmarks from [66] 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, internet of things, and sensor networks [66]. Power domains are specified using the Unified Power Format (UPF) [67], and isolation gates and retention cells are inserted using Synopsys Power Compiler [68] to create a processor implementations that fully support and account for all overheads of power gating. Power analysis is performed using Synopsys Primetime-PX [69]. Experiments were performed on a server housing two Intel Xeon E-2640 processors (8-cores each, 2GHz operating frequency, 64GB RAM).

[5] MSP430 is one of the most popular processors used in low-power systems [22], [63].

TABLE III: Benchmarks

Embedded Sensor Benchmarks [66]
mult, binSearch, tea8, intFilt, div, inSort, rle, tHold, intAVG
EEMBC Embedded Benchmarks [70]
Autocorr, ConvEnc, FFT, Viterbi

TABLE IV: Overheads of implementing power gating (Isolation + Retention)

Domain Type	Module-oblivious			Module-based		
Domain Count	2	3	4	2	3	4
Static Power (%)	1.9	2.5	2.9	0.3	0.7	1.6
Dynamic Power (%)	1.5	3.9	6.0	0.9	1.5	1.6
Area (%)	14.9	18.1	22.2	8.3	8.7	9.7
Wiring (%)	5.1	6.5	11.8	3.4	5.2	3.9
Delay (%)	0.0	0.0	0.0	0.0	0.0	0.0

D. Power Gating Implementation Overheads

Table IV quantifies the overheads of implementing power gating with module-oblivious and module-based domains. Implementation overheads result from insertion of isolation and retention cells. Module-oblivious domains use more isolation cells than module-based domains and thus have higher overhead in terms of power (static and dynamic) and area. Also, module-oblivious domains increase wire length (e.g., for domain control logic), since cells in the same domain may be spread out more across a chip. Despite having higher overhead, module-oblivious domains afford significantly higher power and energy reductions than module-based domains (Section VI). Furthermore, area and wiring overheads did not result in any change in cycle time, as the place and route tool was able to optimize the design for the same timing target. This is not surprising, considering that embedded processors are optimized for low power rather than high performance. Note that area and wiring overheads for module-oblivious domains can be reduced with domain-aware placement and routing optimizations that group cells that belong to the same domain [71]. Such optimizations are beyond the scope of current work.

VI. Results

In this section, we evaluate and analyze the power benefits of module-oblivious power gating compared to aggressive module-based power gating. Note that results for module-based power gating are optimistic, since we allow even software-invisible modules to be power gated. We also do not account for any overhead for hardware-based monitoring logic for module-based power gating. Figure 9 compares the leakage energy savings provided by different power gating techniques. The stacked bars in the figure correspond to three different scenarios. The overall height of a stack shows the potential benefits of the technique when no implementation or instrumentation

overheads are considered, i.e., the maximum potential benefits. The next level in a stacked bar shows benefits after static and dynamic overheads of domain isolation and state retention are accounted for, and the lowest level in a stack shows benefits when accounting for isolation, retention, and software instrumentation overheads. Note that we use the industry-standard UPF format to accurately account for the implementation overheads of power gating. Note also that for our instrumentation overheads, we have conservatively assumed the software approach with the highest overhead – binary instrumentation with dedicated power gating instructions (Section IV-D) – and have accounted for both static and dynamic energy overheads.

Module-oblivious vs. Module-based Domains:

Figure 9 shows that power gating module-oblivious domains can provide significant benefits over conventional module-based domains. On average, power gating on module-oblivious domains provides 1.4× more leakage savings than the maximum savings that can be achieved with module-based domains (Oracle (Module-based)) and 2× more savings than hardware-based management of module-based domains.[6] Figure 10 provides a visualization that explains why module-oblivious domains provide more opportunities for power gating than module-based domains. The figure is a type of correlation matrix that shows the *power gating correlation* between different sub-module pairs (sub-module$_1$, sub-module$_2$) in the processor,[7] where power gating correlation is defined as the fraction of cycles that the two sub-modules, sub-module$_1$ and sub-module$_2$, are both idle at the same time. We have defined the color scale such that cooler colors mean that the sub-modules are more frequently idle at the same time and therefore can be power gated together.

Figure 10a shows the power gating correlation for module-based domains, and Figure 10b is for module-oblivious domains. The different sub-modules in the two matrices are arranged such that sub-modules belonging to the same domain form adjacent rows and columns. The dashed boxes along the main diagonal encircle all the power gating correlation scores for pairs of sub-modules that belong to the same power domain (Figure 4 shows the composition of each module-oblivious domain, and Figure 8 shows the composition of each module-based domain).

Module-based domains do not account for the fact that different parts of the same microarchitectural modules may have uncorrelated activity profiles; as a result, they provide fewer opportunities for power gating. A single sub-module (even a single gate!) with high activity or uncorrelated idle times can sabotage power gating opportunities for an entire domain. For example, even though large portions of the domains in Figure 10a are "cool", the small number of "hot" cells in each domain prevent many power gating opportunities for the domains. Figure 10a shows that in many cases, moving a small number of gates to a different domain could provide more opportunities for power gating *larger areas of logic* for *longer periods of time*. This explains the significant improvement in benefits seen in Figure 9 for module-oblivious power gating over module-based power gating. By forming domains that contain logic from different modules with similar activity profiles, module-oblivious domains do not allow more active logic to ruin power gating opportunities for less active logic in the same module.

Managing Module-oblivious Domains:

While module-oblivious domains provide significant potential for power benefits, they cannot be managed by conventional software- or hardware-based management techniques (Section III). Figure 9 compares the benefits of the proposed software-hardware co-analysis technique for managing module-oblivious domains, which we refer to hereafter as co-analysis, against oracular management. Oracular management assumes perfect knowledge of inputs to make optimal management decisions that exploit every possible cycle of profitable power gating. Co-analysis, on the other hand, uses Xs for inputs to guarantee that power gating decisions will be safe for all possible inputs, since actual inputs are not known at compile time, when co-analysis is performed. Also, since inputs can affect the control paths taken through a program, co-analysis only decides to power gate at a specific point (static instruction) in a program when a domain will not be activated by any possible control path flowing through that point. This ensures safety under all scenarios, even input-dependent data and control. In spite of this conservative approach, results show that co-analysis is a very effective management technique for module-oblivious domains, as it's power benefits are within 8% of optimal (oracle) management of module-oblivious domains.

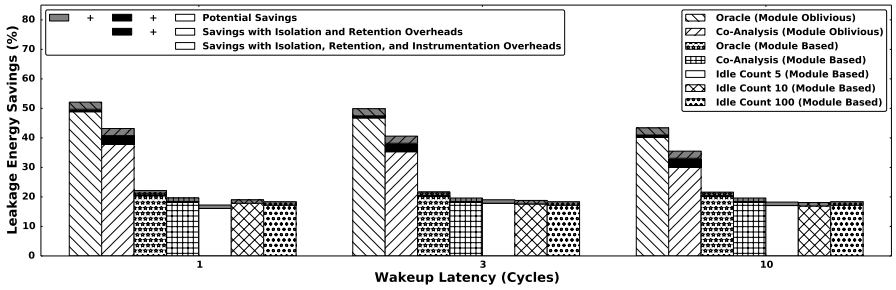
Co-analysis can be used to generate domain management decisions for any set of power domains, no matter how they are formed. We evaluate co-analysis also for module-based domains and compare the benefits achieved against those achieved by state-of-the-art hardware-based management (IdleCount). Figure 9 shows that co-analysis can save (12%) *more* energy than hardware-based domain management for module-based domains, even though we assume no hardware overhead for implementing IdleCount. In fact, the benefits of co-analysis are within 6% of optimal (oracle) for module-based domains. Co-analysis has an advantage over hardware-based domain management even for module-based domains, since co-analysis uses application information to create a tailored power gating strategy for each application, whereas a hardware-based technique necessarily uses the same hard-wired power gating strategy for all applications. Since hardware-based management techniques must apply the same strategy to different applications or application phases that may have different patterns of activity and idleness, they may miss opportunities when power gating is applied too conservatively or incur overheads when power gating is applied too aggressively. For example, a domain managed by IdleCount necessarily spends a fraction of a profitable idle period powered up, as it counts idle cycles before deciding to power down. Also, if a power domain goes to sleep but is needed by an application in fewer cycles than its wakeup latency, the energy penalty for wakeup can cause negative energy savings. In short, the hardware controller for IdleCount must guess the length of idle periods without knowing whether they will be longer than the break-even point. Co-analysis, on the other hand, makes application-aware annotation decisions that account for the break-even point, so it can apply power gating aggressively without ever causing negative energy savings. Finally, note that our implementation is for 65nm technology (Section V), where the leakage power is only 29% of total power. The benefits of our technique are expected to increase for lower (planar) technology nodes, where the problem of leakage power increases significantly [7], [11].

Sensitivity Analysis:

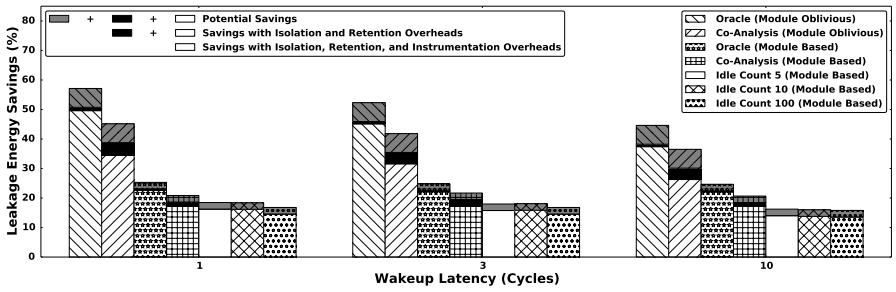
We also compare the power gating techniques across different numbers of power domains and different domain wakeup latencies. The sub-figures of Figure 9 compare the leakage reduction benefits of different power gating techniques for different numbers of domains. Since more domains imply increased specialization of each domain and better adaptation to the activity profile of an application, the potential benefits of power gating generally increase as the number of domains increases. However, our analysis shows that benefits vary by only a few percent for two, three, or four domains. The reason for this behavior is that most of the benefits provided by power gating come from powering down logic with long, correlated idle periods. As illustrated in Figure 10, module-oblivious domains enhance such

[6]Note that these results correspond to full-fledged UPF implementations of power gating that account for all implementation overheads.

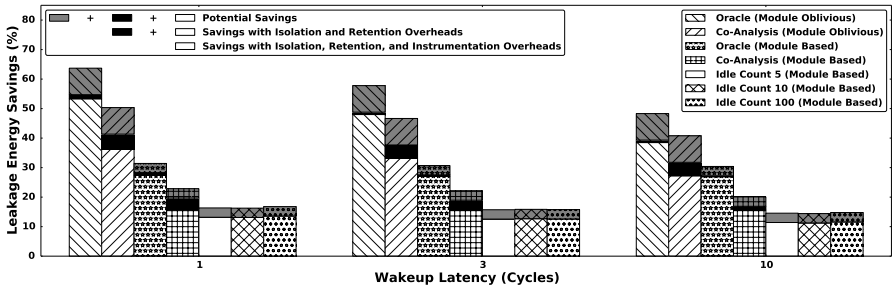
[7]Each sub-module corresponds to one of the module partitions represented as pie sections in Figure 4.



(a) Two Power Domains

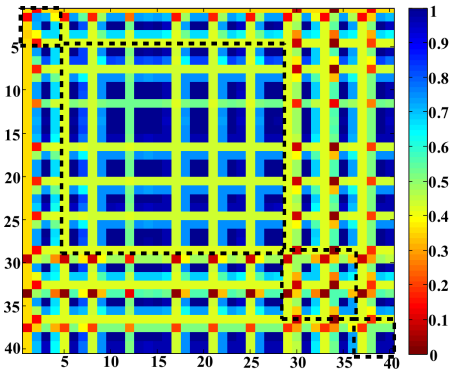


(b) Three Power Domains

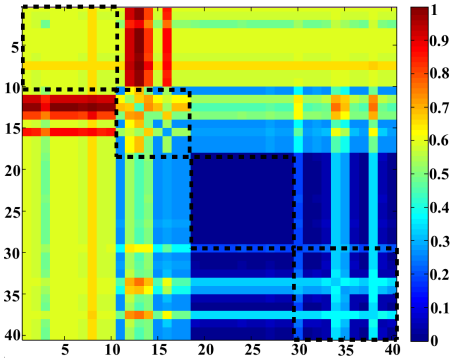


(c) Four Power Domains

Fig. 9: Comparison of leakage energy savings provided by different domain management and formation techniques for different numbers of power domains. Results in each stack (from top to bottom) correspond to maximum potential benefits of the technique, benefits after accounting for isolation and retention overheads, and benefits after accounting for isolation, retention, and instrumentation overheads.



(a) module-based domains



(b) module-oblivious domains

Fig. 10: These "cool" maps compare the potential for power gating between module-based and module-oblivious domains. Cooler colors represent that the logic in a domain is idle together and has more potential to be power gated.

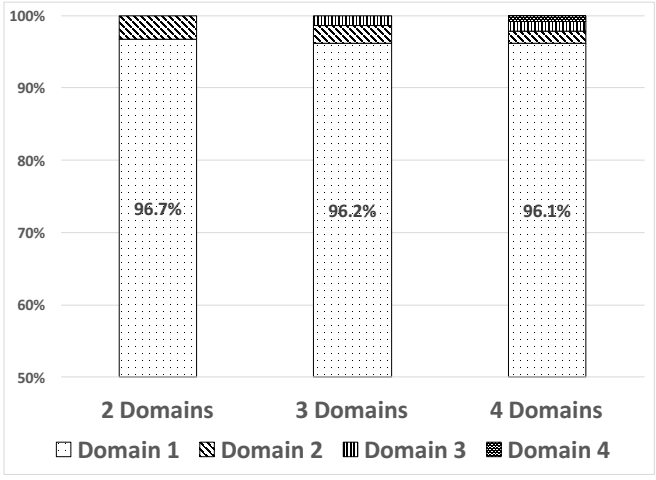


Fig. 11: Most of the energy savings provided by module-oblivious domains are contributed by only a small number of mostly-idle domains.

TABLE V: Performance Overhead (%) introduced by different power domain management techniques.

Domains	Wakeup	IC 5	IC 10	IC 100	Co-analysis
2	1	2.43	2.26	0.23	3.90
	3	4.86	4.52	0.47	3.62
	10	13.38	12.40	1.31	3.40
3	1	3.19	2.35	0.23	6.37
	3	6.39	4.71	0.47	4.67
	10	17.59	12.96	1.31	3.78
4	1	9.18	2.35	0.23	6.59
	3	18.36	4.71	0.47	4.87
	10	50.49	12.96	1.31	3.95

power gating opportunities by relocating logic that would sabotage power gating opportunities for a domain to a different domain with more correlated behavior. This has the effect of collecting logic from different modules into one or more domains that are almost always off. Since only two domains are needed to divide logic into mostly-off and mostly-on domains, only two module-oblivious domains are needed to achieve most of the benefits that they can provide. Figure 11 illustrates this point by showing the percentage of energy savings that each module-oblivious domain contributes to the total. For two, three, and four domains, only one of the domains contributes the majority (96%) of savings (the coolest domain in Figure 10b). This motivates a design with only two domains, since domain isolation and management costs are lower for fewer domains. Module-based domains show similar behavior; however, the presence of logic with uncorrelated activity within some modules limits the length of idle periods and consequently the benefits that can be achieved with module-based domains.

The different clusters of bars within each sub-figure of Figure 9 compare the leakage reduction benefits of different power gating approaches for different wakeup latencies (1, 3, and 10 cycles).[8] As a result, potential benefits decrease with longer wakeup latencies. Shorter wakeup latencies allow power gating to be applied aggressively for shorter idle periods, but this may increase instrumentation overhead due to frequent powering down and up of domains. Our application-aware co-analysis approach accounts for instrumentation overhead and wakeup latency during binary annotation to ensure that a power domain is only powered down when the net effect reduces energy. On average, the time between consecutive power gating decisions is 98 cycles, while the minimum and maximum times between decisions are 5 and 8127 cycles, respectively, demonstrating that correlated activity across gates in different modules often exists at a coarse time granularity. Table V characterizes the performance impact of instrumentation overhead for different numbers of domains and different wakeup latencies, and compares against the performance overheads introduced by hardware-based domain management (IdleCount). In the table, we present the overhead of our most costly binary instrumentation technique (inserting power gating instructions) under the column titled 'Co-analysis'. Since inserting power gating instructions increases runtime, the column represents both performance and energy overhead. Results show that co-analysis has lower performance overhead than IdleCount for low idle count thresholds. While IdleCount has slightly lower performance overhead than co-analysis for a large idle count threshold (100), it loses 100 cycles of potential power gating opportunity for each idle period. In any case, the leakage savings of co-analysis significantly outweigh those of IdleCount (Figure 9), since the hardware-based technique cannot be used to manage module-oblivious domains without incurring prohibitively large implementation overheads.

VII. Conclusions

In this paper, we showed that module-oblivious power gating can provide significantly more leakage savings than state-of-the-art aggressive module-based power gating by allowing larger areas of a processor to be powered down for longer periods of time. Since conventional software- and hardware-based management techniques cannot be applied for module-oblivious power gating, we presented a novel technique for low-cost management of module-oblivious power domains in embedded processors, based on input-independent hardware-software co-analysis. Our technique is automated, does not require programmer intervention, and incurs low management overhead. We demonstrated that module-oblivious power gating based on our technique reduces leakage energy by 2× with respect to state-of-the-art aggressive module-based power gating for a common embedded processor. Our technique for management of module-oblivious power domains achieves leakage energy savings that are within 8% of those achieved by optimal oracular management. Finally, our technique for managing module-oblivious domains is effective even at managing conventional module-based domains. It saves 12% more energy than an idealized implementation of IdleCount – a hardware-based domain management technique for module-based domains. Our benefits are within 6% of optimal for module-based domains.

[8]The domain wakeup latency of 1 cycle is the most realistic for our small embedded processor [72], [29]. We also evaluated 100- and 1000-cycle wakeup latencies in our sensitivity analysis; however, we omitted the results since they showed the same trend as 10-cycle results. Since, it is only profitable to power down a domain if it will be idle for longer than the wakeup latency, wakeup latency has the effect of a low-pass filter on domain power-down/up decisions during analysis. I.e., a given wakeup latency filters out idle periods that are shorter than the wakeup latency.

Acknowledgements

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 also like to thank anonymous reviewers, Arindam Banerjee and Seokhyeong Kang for their feedback.

References

[1] Adam Dunkels, Joakim Eriksson, Niclas Finne, Fredrik Osterlind, Nicolas Tsiftes, Julien Abeillé, and Mathilde Durvy. Low-Power IPv6 for the internet of things. In *Networked Sensing Systems (INSS), 2012 Ninth International Conference on*, pages 1–6. IEEE, 2012.

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

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

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

[5] Ross Yu and Thomas Watteyne. Reliable, Low Power Wireless Sensor Networks for the Internet of Things: Making Wireless Sensors as Accessible as Web Servers. *Linear Technology*, 2013.

[6] G. Hackmann, Weijun Guo, Guirong Yan, Zhuoxiong Sun, Chenyang Lu, and S. Dyke. Cyber-Physical Codesign of Distributed Structural Health Monitoring with Wireless Sensor Networks. *Parallel and Distributed Systems, IEEE Transactions on*, 25(1):63–72, Jan 2014.

[7] International technology roadmap for semiconductors 2.0 2015 edition executive report. Technical report.

[8] Henry Blodget, Marcelo Ballve, Tony Danova, Cooper Smith, John Heggestuen, Mark Hoelzel, Emily Adler, Cale Weissman, Hope King, Nicholas Quah, John Greenough, and Jessica Smith. The internet of everything: 2015. *BI Intelligence*, 2014.

[9] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. April 2011.

[10] Gil Press. Internet of Things By The Numbers: Market Estimates And Forecasts. *Forbes*, 2014.

[11] Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003.

[12] Danbee Park, Jungseob Lee, Nam Sung Kim, and Taewhan Kim. Optimal algorithm for profile-based power gating: A compiler technique for reducing leakage on execution units in microprocessors. In *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, pages 361–364, Nov 2010.

[13] Vasileios Kontorinis, Amirali Shayan, Dean M. Tullsen, and Rakesh Kumar. Reducing Peak Power with a Table-driven Adaptive Processor Core. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 189–200, New York, NY, USA, 2009. ACM.

[14] Paula Petrica, Adam M. Izraelevitz, David H. Albonesi, and Christine A. Shoemaker. Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems. *SIGARCH Comput. Archit. News*, 41(3):13–23, June 2013.

[15] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[16] A Ahmed and Wayne Wolf. Hardware/software interface codesign for embedded systems. 2005.

[17] ARM. Internet of Things IoT.

[18] Leo Sun. ARM Holdings PLC Dives Deeper Into the Internet of Things.

[19] National Instruments. Compile Faster with the LabVIEW FPGA Compile Cloud Service.

[20] Cloud Compiling. Cloud Compiling.

[21] ARM. ARM mbed IoT Device Platform.

[22] Jacob Borgeson. Ultra-low-power pioneers: TI slashes total MCU power by 50 percent with new "Wolverine" MCU platform. *Texas Instruments White Paper*, 2012.

[23] ARM. Cortex-A9 Technical Reference Manual. *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc .ddi0388e/index.html*.

[24] ARM. ARM Cortex-A15 MPCore Processor Technical Reference Manual. *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc .ddi0438i/CJHEAECF.html*.

[25] Atmel. AT06549: Ultra Low Power Techniques. *http://www.atmel.com/images/atmel-42411-ultra-low-power-techniques-at06549_application-note.pdf*.

[26] Intel. The Power Management IC for the Intel Atom Processor E6xx Series and Intel Platform Controller Hub EG20T. *http://www.intel.com/content/dam/www/public/us/en /documents/white-papers/atom-e6xx-power-management-ic-paper.pdf*.

[27] Intel. Dynamic Power Gating Implementation on Intel Embedded Media and Graphics Driver. *http://www.intel.com/content/dam/www/public/us/en /documents/white-papers/emgd-dynamic-power-gating-paper.pdf*.

[28] Zhigang Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 32–37, Aug 2004.

[29] Anita Lungu, Pradip Bose, Alper Buyuktosunoglu, and Daniel J. Sorin. Dynamic Power Gating with Quality Guarantees. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, pages 377–382, New York, NY, USA, 2009. ACM.

[30] Youngsoo Shin, Jun Seomun, Kyu-Myung Choi, and Takayasu Sakurai. Power Gating: Circuits, Design Methodologies, and Best Practice for Standard-cell VLSI Designs. *ACM Trans. Des. Autom. Electron. Syst.*, 15(4):28:1–28:37, October 2010.

[31] H. Tabkhi and G. Schirner. Application-Guided Power Gating Reducing Register File Static Power. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(12):2513–2526, Dec 2014.

[32] J. Kao, S. Narendra, and A. Chandrakasan. MTCMOS hierarchical sizing based on mutual exclusive discharge patterns. In *Design Automation Conference, 1998. Proceedings*, pages 495–500, June 1998.

[33] Mohab Anis, Mohamed Mahmoud, Mohamed Elmasry, and Shawki Areibi. Dynamic and leakage power reduction in mtcmos circuits using an automated efficient gate clustering technique. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 480–485, New York, NY, USA, 2002. ACM.

[34] Changbo Long and Lei He. Distributed Sleep Transistor Network for Power Reduction. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 181–186, New York, NY, USA, 2003. ACM.

[35] A. Abdollahi, F. Fallah, and M. Pedram. An effective power mode transition technique in mtcmos circuits. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 37–42, June 2005.

[36] Kimiyoshi Usami and Naoaki Ohkubo. A design approach for fine-grained run-time power gating using locally extracted sleep signals. In *Proc. of ICCD'06*, pages 155–161, 2006.

[37] Ashoka Sathanur, Antonio Pullini, Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Timing-driven Row-based Power Gating. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ISLPED '07, pages 104–109, New York, NY, USA, 2007. ACM.

[38] Y. Kanno, H. Mizuno, Y. Yasu, K. Hirose, Y. Shimazaki, T. Hoshi, Y. Miyairi, T. Ishii, Tetsuy. Yamada, T. Irita, T. Hattori, K. Yanagisawa, and N. Irie. Hierarchical Power Distribution With Power Tree in Dozens of Power Domains for 90-nm Low-Power Multi-CPU SoCs. *Solid-State Circuits, IEEE Journal of*, 42(1):74–83, Jan 2007.

[39] Tong Xu, Peng Li, and Boyuan Yan. Decoupling for power gating: Sources of power noise and design strategies. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 1002–1007, June 2011.

[40] De-Shiuan Chiou, Da-Cheng Juan, Yu-Ting Chen, and Shih-Chieh Chang. Fine-Grained Sleep Transistor Sizing Algorithm for Leakage Power Minimization. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 81–86, June 2007.

[41] B.H. Calhoun, F.A. Honore, and A. Chandrakasan. Design methodology for fine-grained leakage control in MTCMOS. In *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pages 104–109, Aug 2003.

[42] Abhinav Agarwal and Arvind. Leveraging Rule-based Designs for Automatic Power Domain Partitioning. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 326–333, Piscataway, NJ, USA, 2013. IEEE Press.

[43] Lizhong Chen and Timothy M Pinkston. Nord: Node-router decoupling for effective power-gating of on-chip routers. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 270–281. IEEE Computer Society, 2012.

[44] Reetuparna Das, Satish Narayanasamy, Sudhir K Satpathy, and Ronald G Dreslinski. Catnap: energy proportional multiple network-on-chip. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 320–331. ACM, 2013.

[45] Lizhong Chen, Di Zhu, Massoud Pedram, and Timothy M Pinkston. Power punch: Towards non-blocking power-gating of noc routers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 378–389. IEEE, 2015.

[46] Danbee Park, Jungseob Lee, Nam Sung Kim, and Taewhan Kim. Optimal algorithm for profile-based power gating: A compiler technique for reducing leakage on execution units in microprocessors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 361–364. IEEE Press, 2010.

[47] Randal E Bryant. Symbolic simulationtechniques and applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 517–521. ACM, 1991.

[48] A. Kolbi, J. Kukula, and R. Damiano. Symbolic RTL simulation. In *Design Automation Conference, 2001. Proceedings*, pages 47–52, 2001.

[49] Tao Feng, L. C. Wang, Kwang-Ting Cheng, M. Pandey, and M. S. Abadir. Enhanced symbolic simulation for efficient verification of embedded array systems. In *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, pages 302–307, Jan 2003.

[50] P. Jain and G. Gopalakrishnan. 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, Aug 1994.

[51] L. Liu and S. Vasudevan. Efficient validation input generation in RTL by hybridized source code analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

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

[53] Y. Zhang, Z. Chen, and J. Wang. Speculative symbolic execution. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 101–110, Nov 2012.

[54] O Girard. OpenMSP430 project. *available at opencores.org*, 2013.

[55] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[56] K. Hamaguchi. Symbolic simulation heuristics for high-level design descriptions with uninterpreted functions. In *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, pages 25–30, 2001.

[57] Steven Cherry. Hacking Pacemakers.

[58] Wikipedia. Bare Machine, Wikipedia.

[59] Texas Instruments. StarterWare.

[60] ARM. Building BareMetal ARM systems with GNU.

[61] Lior Rokach and Oded Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.

[62] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 32–37. ACM, 2004.

[63] Wikipedia. List of wireless sensor nodes, 2016. [Online; accessed 7-April-2016].

[64] Synopsys. Design Compiler User Guide. *http://www.synopsys.com/*.

[65] Cadence. Encounter Digital Implementation User Guide. *http://www.cadence.com/*.

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

[67] IEEE Standard for Design and Verification of Low-Power Integrated Circuits. *IEEE Std 1801-2013 (Revision of IEEE Std 1801-2009)*, pages 1–348, May 2013.

[68] Synopsys. Power Compiler User Guide. *http://www.synopsys.com/*.

[69] Synopsys. PrimeTime User Guide. *http://www.synopsys.com/*.

[70] EEMBC, Embedded Microprocessor Benchmark Consortium.

[71] Hailin Jiang and Malgorzata Marek-Sadowska. Power-gating aware floorplanning. In *Quality Electronic Design, 2007. ISQED'07. 8th International Symposium on*, pages 853–860. IEEE, 2007.

[72] P. Royannez, H. Mair, F. Dahan, M. Wagner, M. Streeter, L. Bouetel, J. Blasquez, H. Clasen, G. Semino, J. Dong, D. Scott, B. Pitts, C. Raibaut, and Uming Ko. 90nm low leakage soc design techniques for wireless applications. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 138–589 Vol. 1, Feb 2005.