Application-specific Design and Optimization for Ultra-Low-Power Embedded Systems

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Hari Cherupalli

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

John M Sartori

August, 2019

Acknowledgements

This journey of PhD would not have been possible without my advisor Prof. John Sartori. I will forever be grateful for his support and guidance during my PhD. I am also grateful to him for giving me the opportunity and freedom to pursue the problems I was interested in and for this patience in guiding me when I needed help. His encouraging and kind nature allowed me to get through the stressful part of graduate school and enjoy my research. I would also like to express my graditude to the National Science Foundation for supporting my research through NSF grant #1255861, to the College of Science and Engineering for supporting my first year through a Fellowship and to the University of Minnesota for supporting my final year through the Doctoral Dissertation Fellowship. I cannot thank Prof. Rakesh Kumar, from the University of Illinois, enough for his guidance throughout my PhD. He complemented the liberty John has given me with providing structure and direction to my research, and also demonstrated the importance of good collaborations.

I would like to acknowledge my collaborators for their contributions to my research. Particularly, I would like to thank Henry Duwe for believing that we can actually pull off the last paper and really helping me through it. I would like to thank Weidong Ye for always being there when I needed help with getting simulations done and especially for his expertise in generating plots in time. I would also like to thank Himanshu Sahoo for helping generate results on a real processor.

The time I spent with my friends and labmates typifies my experience in grad school. I would like to express my gratitude to Manas Minglani, Sriharsha Vadlamani, Saroj Sarapathy, Amogha Gundavaram, and Veena Kondaveeti for being there during stressful situations. My friends Oscar Rodriguez, Kartik Ramkrishnan, Mohit Sinha, Abhishek Agrawal, Saurabh Chaubey, Naveen Elangovan, Min Shi, Vishal Jamkar and Deepanjan

Panda made my time memorable and enjoyable.

Most of my accomplishments would not have been possible without the support of my parents. The sacrifices they made for my upbringing and the values they instilled have given me the motivation and strength I needed.

Dedication

To my father for instilling the thirst for knowledge in me.

Abstract

The last few decades have seen a tremendous amount of innovation in computer system design to the point where electronic devices have become very inexpensive. This has brought us on the verge of a new paradigm in computing where there will be hundreds of devices in a person's environment, ranging from mobile phones to smart home devices to wearables to implantables, all *interconnected*. This paradigm, called the Internet of Things (IoT), brings new challenges in terms of power, cost, and security.

For example, power and energy have become critical design constraints that not only affect the lifetime of an ultra-low-power (ULP) system, but also its size and weight. While many conventional techniques exist that are aimed at energy reduction or that improve energy efficiency, they do so at the cost of performance. As such, their impact is limited in circumstances where energy is very constrained or where significant degradation of performance or functionality is unacceptable. Focusing on the opposing demands to increase both energy efficiency and performance simultaneously in a world where Moore's law scaling is decelerating, one of the underlying themes of this work has been to identify novel insights that enable new pathways to energy efficiency in computing systems while avoiding the conventional tradeoff that simply sacrifices performance and functionality for energy efficiency.

To this end, this work proposes a method to analyze the behavior of an application on the gate-level netlist of a processor for all possible inputs using a novel symbolic hardware-software co-analysis methdology. Using this methodology several techniques have been proposed to optimize a given processor-application pair for power, area and security.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A large number of existing and emerging computing applications have ultra-low-power requirements [4, 5, 6, 7, 8]. Notable among these are the internet of things, sensor networks, wearable electronics, and biomedical devices. The low-power requirements of these applications are due to the fact that the applications are either energy-constrained (e.g., battery-powered applications), power-constrained (e.g., energy-harvesting applications), or both. These applications rely on low-power microcontrollers and microprocessors that have become the most widely-used type of processor in production [9, 10, 11]. In the low-power embedded systems used by these applications, energy efficiency is the primary factor that determines critical system characteristics such as size, weight, cost, reliability, and lifetime [12, 13].

One option to target these applications is to use application-specific integrated circuits (ASICs), given their energy efficiency advantages over general purpose processors (GPPs). However, GPPs are the preferred solution for many low-power applications, due to their evolving nature as well as the high costs of custom IC design. Existing power management techniques for GPPs trade off performance to reduce power. However, power can only be reduced to the extent where the associated performance reduction is acceptable; existing power management techniques offer no solution for power reduction beyond the point where performance reduction becomes unacceptable. Given the stringent power and energy constraints of emerging and existing low-power systems, solutions that sacrifice performance to reduce power may be unacceptable. Therefore, one of the primary focuses of this work has been to reduce power without reducing performance.

These novel opportunities are based on the observation that resources that an application does not use do not contribute to application performance. Therefore, any power expended by resources that an application does not use can theoretically be eliminated, bringing the power consumption of a GPP running an application closer to that of an ASIC. To exploit this opportunity, this work proposes a novel *hardware-software co-analysis* technique that determines the maximal set of hardware resources that an application can use during execution, irrespective of application inputs. We then use the input-independent activity profile of an application running on a GPP to identify various opportunities to optimize the processor-application pair for power, cost and security. Specifically, we propose the following optimizations.

Application-specific Timing Analysis: Using co-analysis, we will determine the longest path exercised by an application. If the application-specific longest path is shorter than the worst-case longest path of the design (i.e., the static critical path), we can determine a lower-than-nominal minimum operating voltage that is guaranteed to be safe for the application.

Application-specific Power Gating: We can use gate activity profiles to both group gates into application-specific power domains and make application-specific power gating decisions. Application-specific power gating can provide opportunities to power gate larger areas of logic for longer periods of time than state-of-the-art power gating techniques.

Application-specific Peak Power Management: Co-analysis enables us to guarantee application-specific bounds on the peak power and energy requirements of a GPP. This allows us to determine the minimum application-specific guardband against voltage variations. For battery-powered and energy harvesting systems, this also enables application-specific sizing for the battery and harvester, which can be significantly smaller compared to application-oblivious sizing.

Application-specific Processor Pruning (Bespoke Processors): Based on the information provided by the gate activity profiles, we can infer which gates will not be used by a given application. Knowledge of which gates cannot be toggled can be used to produce a trimmed 'bespoke' processor that has a lower area and cost. In settings where the costs of fabricating a processor are significantly reduced, such as printed plastic and organic processors that can be 3D printed, reducing costs to produce application specific designs using co-analysis can be significantly valuable.

Application-specific Gate-level Information Flow Tracking: Augmenting co-analysis with information flow tracking symbols where each net is marked as 'trusted' or 'untrusted' allows us to perform application-specific information flow tracking for a processor-application pair. This software-based solution to gate-level information flow tracking identifies all possible insecure information flows for the software running on a system, as well as the instructions that can cause violations, and allows security vulnerabilities to be eliminated through software modifications. By targeting only the vulnerabilities that an application is susceptible to, we minimize, and in some cases completely eliminate, the overhead of guaranteeing security for ULP systems

- Chapter 2 presents the proposed symbolic hardware software co-analysis technique and subsequent chapters present various applications of the technique.
- In Chapter 3 describes how hardware-software co-analysis can be used to exploit dynamic timing slack for power savings.
- Chapter 4 describes how hardware-software co-analysis can enable effective module-oblivious power domain construction and management for greater leakage energy savings.
- Chapter 5 describes the application of the hardware-software co-analysis technique to estimate peak power and energy of a given application-processor pair.
- Chapter 6 describes another application of the technique to produce an application-specific bespoke processor for a given application.
- Chapter 7 describes how the hardware-software co-analysis technique can be used to validate a processor for information flow security at the gate-level and proposes software techniques to ensure information flow security for embedded processors.
- In Chapter 8 concludes the thesis with discussion of future directions.

Chapter 2

Hardware-Software Co-Analysis

One of the key observations made in this research is that for many of the emerging applications in the IoT space **it is possible to have the complete hardware and software specification of the system.** With the growing movement towards open-source hardware, this is getting more and more possible. Also, with the movement towards a cloud service model of providing software or design information through a secure portal this is possible. We leverage this observation in this research not only to improve energy efficiency but also to reduce design and implementation costs and improve the security of ultra-low-power systems.

Exploiting the complete knowledge of the hardware and software specification of an embedded processor-application pair, we have developed a methodology that can efficiently analyze the behavior of an application on the gate-level netlist of a processor, for all possible executions of the application. Using this analysis, one can identify all parts of a processor that the application is guaranteed to not exercise. To capture the behavior of the application for all possible executions, we symbolically simulate the application binary on the gate-level netlist of the processor, where inputs to the application are represented using symbols representing unknown logic values. By tracking these symbols and the gate-level activity of the design, one can identify what functionalities of a processor an application is guaranteed not to use at any time granularity from a per-cycle basis up to an entire application.

Symbolic simulation has been applied in circuits for logic and timing verification, as well as sequential test generation [14, 15, 16, 17, 18, 19]. Symbolic simulation has also been applied for software verification [20]. However, to the best of our knowledge, symbolic co-analysis of software and hardware at the granularity of the gate-level has never been studied. Using the information gathered during this co-analysis, we proposed several novel optimizations both at the hardware and software level for power savings, size and cost reduction, and security in ultra-low-power systems.

Hardware-Software Co-analysis

The set of gates that an application toggles during execution can vary depending on application inputs. This is because inputs can change the control flow of execution through the code as well as the data paths exercised by the instructions. Since exhaustive profiling for all possible inputs is infeasible, and limited profiling may not identify all exercisable gates in a processor, we have implemented an analysis technique based on symbolic simulation [21], that is able to characterize the gate-level activity of a processor executing an application for all possible inputs with a single gate-level simulation. During this simulation, inputs are represented as unknown logic values (Xs), which are treated as both 1s and 0s when recording possible toggled gates.

Algorithm 1 describes input-independent gate activity analysis. Initially, the values of all memory cells and gates are set to Xs. The application binary is loaded into program memory, providing the values that effectively constrain which gates can be toggled during execution. During simulation, our simulator sets all inputs to Xs, which propagate through the gate-level netlist during simulation.[1] After each cycle is simulated, the toggled gates are removed from the list of unexercisable gates. Gates where an X propagated are considered as toggled, since some input assignment could cause the gates to toggle. If an X propagates to the PC, indicating input-dependent control flow, our simulator branches the execution tree and simulates execution for all possible branch paths, following a depth-first ordering of the control flow graph. Since this naive simulation approach does not scale well for complex or infinite control structures which result in a large number of branches to explore, we employ a conservative approximation that allows our analysis to scale for arbitrarily-complex control structures while

[1] Any data or signals that can be written by external events (e.g., interrupt signals or DMA writes) are also considered unknown values (Xs) during our analysis. Firmware components of interrupt handling, e.g., the jump table and interrupt service handling routine, are considered to be part of the application binary (i.e., known values) during symbolic simulation. If an interrupt is enabled during an instruction's execution, then that instruction is considered as possibly modifying the PC.

conservatively maintaining correctness in identifying exercisable gates. Our approximation works by tracking the most conservative gate-level state that has been observed for each PC-changing instruction (e.g., conditional branch). The most conservative state is the one where the most variables are assumed to be unknown (X). When a branch is re-encountered while simulating on a control flow path, simulation down that path can be terminated if the symbolic state being simulated is a substate of the most conservative state previously observed at the branch (i.e., the states match or the more conservative state has Xs in all differing variables), since the state (or a more conservative version) has already been explored. If the simulated state is not a substate of the most conservative observed state, the two states are merged to create a new conservative symbolic state by replacing differing state variables with Xs, and simulation continues from the conservative state. This conservative approximation technique allows gate activity analysis to complete in a small number of passes through the application code, even for applications with an exponentially-large or infinite number of execution paths.[2]

[2] Some complex applications and processors might still require heuristics for exploration of a large number of execution paths [22, 23]; however, our approach is adequate for ULP systems, representative of an increasing number of future applications which tend to have simple processors and applications [24, 11]. For example, complete analysis of our most complex benchmark takes 3 hours.

Algorithm 1 Input-independent Gate Activity Analysis

```
1: Procedure Annotate Gate-Level Netlist(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: $s \leftarrow$ State at start of app_binary
6: Table of previously observed symbolic states, $T$.insert($s$)
7: Stack of un-processed execution points, $U$.push($s$)
8: mark_all_gates_untoggled(design_netlist)
9: while $U$ != $\emptyset$ do
10:     $e \leftarrow U$.pop()
11:     while $e$.PC_next != X and !$e$.END do
12:         $e$.set_inputs_X() // set all peripheral port inputs to Xs
13:         $e' \leftarrow$ propagate_gate_values($e$) // simulate this cycle
14:         annotate_gate_activity(design_netlist,$e$,$e'$) // unmark every gate toggled (or possibly toggled)
15:         if $e'$.modifies_PC then
16:             $c \leftarrow T$.get_conservative_state($e$)
17:             if $e' \not\subset c$ then
18:                 $T$.make_conservative_superstate($c$,$e'$)
19:             else
20:                 break
21:             end if
22:         end if
23:         $e \leftarrow e'$ // advance cycle state
24:     end while
25:     if $e$.PC_next == X then
26:         $c \leftarrow T$.get_conservative_state($e$)
27:         if $e \not\subset c$ then
28:             $e' \leftarrow T$.make_conservative_superstate($c$,$e$)
29:             for all $a \in$ possible_PC_next_vals($e'$) do
30:                 $e'' \leftarrow e$.update_PC_next($a$)
31:                 $U$.push($e''$)
32:             end for
33:         end if
34:     end if
35: end while
36: for all $g \in$ design_netlist do
37:     if $g$.untoggled then
38:         annotate_constant_value($g$,$s$) // record the gate's initial (and final) value
39:     end if
40: end for
```

Chapter 3

Application-specific Timing Analysis

Hardware-software co-analysis reveals which functionalities in a processor can be exercised by an application and which functionalities will never be used by the application. Through co-analysis, we have observed that the embedded software application running on a low-power processor may not utilize all the functionalities provided by the processor. Only the parts of the processor that can be utilized by the application need to meet timing constraints to ensure that the application executes correctly. Therefore, in scenarios where unused functionalities correspond to timing-critical logic, there may exist timing slack between the most timing-critical functionalities that exist in the processor and the most timing-critical functionalities that are exercised by the embedded software application running on the processor. We call this application-specific timing slack as *dynamic timing slack* (DTS). In this chapter, we will investigate the extent to which application-specific DTS exists for low-power applications and processors and can be exploited to improve their energy efficiency.

3.1 Workload-Dependent Dynamic Timing Slack

Most modern processors are synchronous or clocked. This means that computation is performed in clock periods, where data is transmitted from a launch flip-flop (FF) to a capture FF (endpoint) through combinational logic gates. Transmission from launch to capture FF begins with a clock signal to the launch FF and must complete before the next clock signal reaches the capture FF (i.e., one clock period).[1] For example, a logic transition (toggle) of the path in Figure 3.1 initiated by a toggle at the Q-pin of FF1 (the launch FF) must reach the D-pin of FF2 (the capture FF) in one clock period. A path that respects these constraints is said to *meet timing.* If the combined delay of gates G1 through G4 is greater than the clock period, the path does not meet timing. A typical processor has a large number of paths [25], and traditionally, all paths *must* meet timing. However, some paths may just meet timing with little time to spare (timing-critical paths), while for other paths the correct data arrives at the D-pin of the capture FF (endpoint) significantly before the end of a clock period.

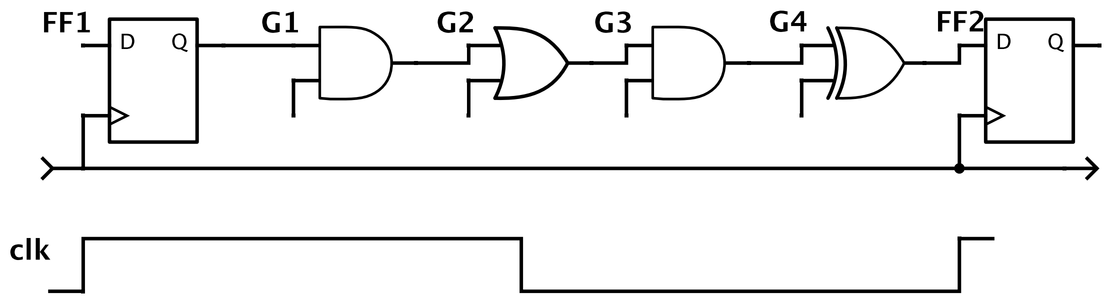


Figure 3.1: A path in a synchronous circuit

Decreasing a processor's operating voltage reduces power but also increases logic delays, which can cause timing-critical paths to violate timing constraints. Note, however, that if the output of a path (e.g., the D-pin of FF2 in Figure 3.1) is never toggled by an application, the capture FF will capture the same (constant) value in each clock period, and the non-toggling path will *produce correct output values even if the path violates timing constraints.* Co-analysis reveals that many applications for low-power processors do not utilize a GPP's entire feature set [5, 7, 6]. Non-exercised features can mean that only a subset of the paths in a processor are exercised (toggled). If the longest *exercised* path in the processor is not a timing-critical path (i.e., it produces data at its endpoint with time to spare), then there exists an opportunity to trade this extra timing slack for reduced power *at no performance cost* by keeping frequency constant and reducing

[1] For simplicity, this discussion omits details such as setup and hold time, guardbands, etc. that are accounted for during timing analysis.

the processor's voltage to the lowest voltage where all *exercised* paths still meet timing. Un-exercised paths are allowed to violate timing constraints. This is the opportunity we exploit in this chapter.

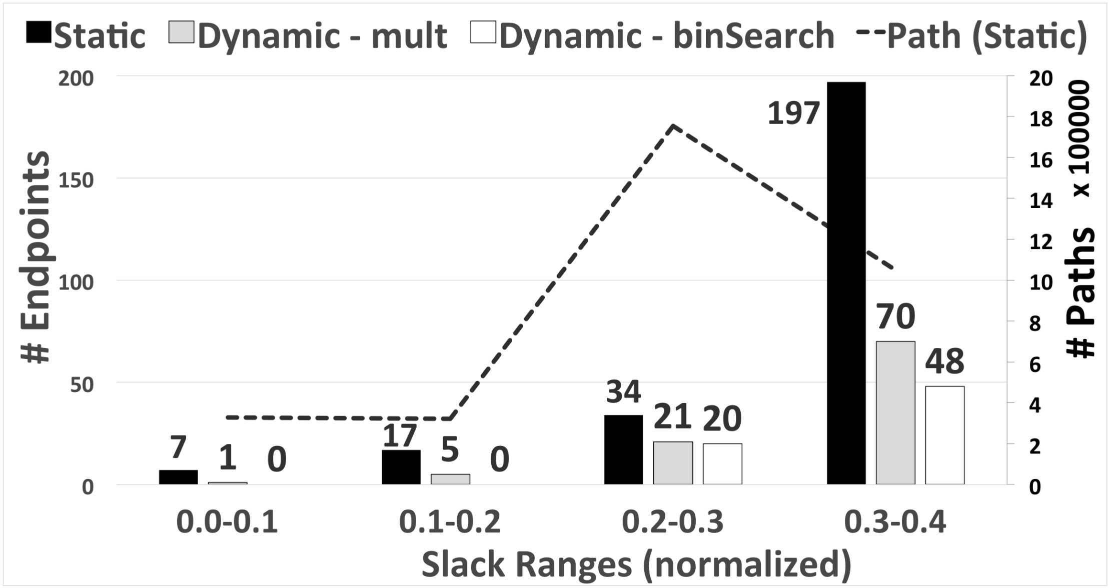


Figure 3.2: The dynamic slack distributions for two applications (*mult* and *binSearch*) on openMSP430 show that both applications do not exercise all of the endpoints in the processor. Slack is normalized to the clock period.

We now provide a motivational example that illustrates the existence of DTS in a low-power processor. Figure 3.2 compares several slack distributions for slack up to 40% of the clock period, for a fully synthesized, placed, and routed openMSP430 processor [26]. In the figure, the x-axis has bins for various slack ranges (normalized to the clock period), the left y-axis shows the number of processor endpoints with worst slack in a particular range, and the right y-axis shows the number of paths with slack in a particular range.[2] The static slack distributions, *Static* and *Path (Static)*, characterize the worst slacks of all endpoints and paths in the processor, respectively, whether

[2] Worst slack is defined for an endpoint (FF) as the timing slack of the longest path terminating at that endpoint. Since many paths lead to the same endpoint, the number of paths in a design is typically several orders of magnitude larger than the number of endpoints [25]. In our processor, each endpoint corresponds to tens or hundreds of thousands of paths.

exercised or not [27, 28, 29]. Note that a large number of paths in the design are statically critical (over 325,000). This is consistent with previous observations on other designs [30, 31]. Nevertheless, when a particular application is executed on the processor, it may not exercise all paths or path endpoints. The other two series in Figure 3.2 show the distributions of worst slacks for only the endpoints in the processor that are exercised by multiplication (*mult*) and binary search (*binSearch*) applications. We call these *dynamic slack distributions*, and we call the longest exercised paths in a design the *dynamic critical paths* for a particular application [32, 33]. Slack distributions are reported at the worst-case (slow) corner to isolate DTS from all other phenomena that might affect the minimum operating voltage for an application (e.g., process, voltage, temperature, or aging variations).

The following observations and inferences can be drawn from Figure 3.2.

• Several endpoints of the processor (and hence, orders of magnitude more paths) are not exercised when a particular application is executed. This is demonstrated by the difference between the static slack distribution and the two dynamic slack distributions. For example, the processor contains seven endpoints (and hundreds of thousands of paths) with worst slack in the range [0.0 - 0.1] and 17 endpoints in the range of [0.1 - 0.2], but *binSearch* does not exercise any of those endpoints (or their associated paths).

• Different applications exercise different processor features and can have different dynamic critical paths. Consequently, the amount of available DTS can be different for different applications. For example, since *binSearch* does not exercise any endpoints with worst slack less than 0.2, its normalized DTS is at least 0.2. On the other hand, *mult* exercises one endpoint with worst slack in the range [0.0 - 0.1] (the multiplier overflow register), and it has less DTS than *binSearch*.

• DTS represents an opportunity to *save power without sacrificing performance*. For example, if *binSearch* is executing on the processor, the operating voltage can be reduced while keeping frequency constant, such that paths with timing slacks of up to 20% of the clock period of the processor violate timing constraints (since these paths are not exercised by the application). This generates power savings without affecting either the functionality or performance of the processor for *binSearch*. Note also that unlike timing speculative approaches that save power by reducing safety guardbands (e.g., Razor [34, 35, 36]), exploiting DTS does not require guardband reduction and therefore is *completely non-speculative.* Exploiting DTS simply involves adjusting the voltage of the processor to the minimum *safe* voltage for the subset of processor logic that is exercised by an application. Guardbands for the exercised logic are not violated. Given that DTS exists for some applications, "free" power savings can be attained at no performance cost and no risk to timing safety by adjusting the operating voltage of the processor to exploit DTS while leaving design guardbands in place.

3.2 Quantifying DTS in Processors

Low-power embedded systems are a promising context for exploiting DTS, since embedded applications typically do not use all of the hardware features provided by a processor. Such applications may, therefore, not exercise the most timing critical logic in a processor [37, 33, 38]. Also, low-power processors are optimized to minimize area and power rather than maximize performance (e.g., many common microcontrollers have a small number of pipeline stages[3]), which typically results in relatively less balanced logic across pipeline stages or more delay variation across processor logic within a pipeline stage. This may increase available DTS, since in a design with larger delay variation, finite options for cell drive strength, threshold voltage, layout, etc. mean that not all paths will become timing-critical after design optimization.[4] Previous research has also observed that only a fraction of logic in an embedded design may be timing-critical [39].

Preliminary measurement-based results provide evidence of DTS in common low-power processors. Experiments were performed for PIC24FJ64GA002 and MSP430F1610 processors by fixing the operating frequency and lowering the supply voltage to observe whether different applications exhibit different minimum safe operating voltages. Table 3.1 reports the minimum voltage at which each application operates without errors, along with the power savings with respect to operation at the nominal voltage. The observed V_{min} varies by up to 120 mV for different applications on PIC24 and by up to 160 mV on MSP430, suggesting the existence of significant DTS.[5]

[3] Many PIC and Atmel microcontrollers have only two stages.

[4] Preliminary results show significant power reduction from exploiting DTS for a processor that was synthesized, placed, and routed using an aggressive industry-standard design methodology that minimizes timing slack as much as possible.

[5] Some fraction of the V_{min} differences across benchmarks in measured results could theoretically be

Table 3.1: Observed V_{min} on PIC24 ($V_{nom} = 2.0V$) and MSP430 ($V_{nom} = 3.3V$) for different sensor network benchmarks [1].

	PIC24		MSP430	
Benchmark	$V_{min}(V)$	Pwr Saved (%)	$V_{min}(V)$	Pwr Saved (%)
binSearch	1.82	20.2	2.87	30.3
div	1.83	20.3	2.87	33.7
inSort	1.85	17.2	2.90	36.2
intAVG	1.89	13.1	2.77	38.4
intFilt	1.83	20.0	2.92	30.5
mult	1.82	20.4	2.76	41.7
rle	1.77	25.5	2.83	35.9
tHold	1.83	20.1	2.86	34.4
tea8	1.82	20.4	2.82	39.5

3.3 Identifying and Exploiting Application-Specific DTS

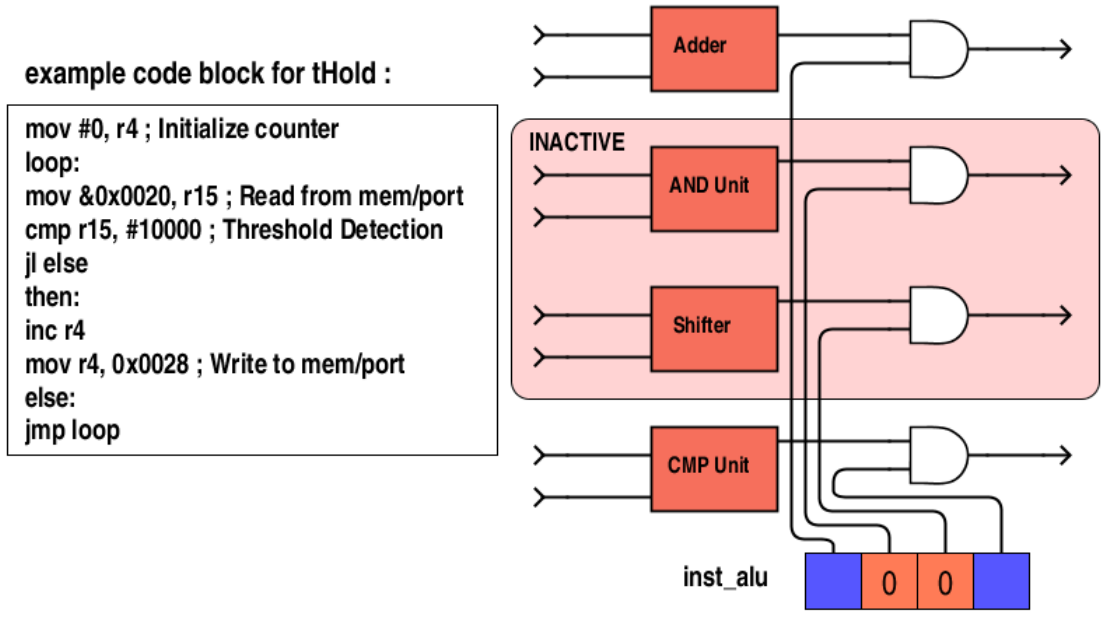
DTS identification is based on the observation that if part of a processor design cannot be exercised by its embedded software application, then it can be constrained to a constant value or ignored during design timing analysis to expose DTS and reveal a more aggressive operating voltage. To expose DTS, the constraints identified during hardware-software co-analysis (i.e., nets in the design that can never be toggled by the application) are applied to the gate-level netlist, and static timing analysis is performed on the constrained design to determine the minimum voltage at which the design is guaranteed to operate safely for the given application. Application-specific timing analysis is performed for worst case process, voltage, and temperature (PVT) conditions such that the minimum operating voltage reported is guaranteed to be safe independent of PVT variations.

All paths that pass through an un-toggled net or gate can be ignored during application-specific timing analysis. Such paths, by definition, are not toggled by the application, and the application will complete successfully even if these paths do not meet timing constraints.

Once all constraints identified by co-analysis have been applied to a design, application-specific timing analysis checks whether all of the exercisable paths remaining in the

due to input-dependent voltage and temperature variations, in spite of ultra-low currents. Preliminary results reported in Section 3.4 isolate the impact of DTS alone, since they are captured assuming worst case variations and inputs.

design (e.g., the dynamic critical paths) meet timing constraints. The minimum safe operating voltage for the constrained design can be determined by lowering the voltage in steps and performing constrained timing analysis at each step to find the lowest voltage at which all paths in the constrained design meet timing constraints.



```
mov #0, r4 ; Initialize counter
loop:
mov &0x0020, r15 ; Read from mem/port
cmp r15, #10000 ; Threshold Detection
jl else
then:
inc r4
mov r4, 0x0028 ; Write to mem/port
else:
jmp loop
```

Figure 3.3: DTS identification enabled by hardware-software co-analysis.

For an example of how co-analysis can automatically identify un-exercised logic in a design that can be constrained to expose DTS, consider the `inst_alu` register in open-MSP430. This 12-bit one-hot encoded register selects the function unit that will execute an instruction. A bit that selects a particular function unit is set by an instruction that executes on the function unit. Not all applications utilize the entire instruction set, and a bit in `inst_alu` will not be toggled by an application that does not use the function unit selected by the bit. For example, a run length encoding (*rle*) application we evaluated does not use right shift or left shift instructions. Thus, the select bits corresponding to these instructions' function units remain as constant zeros during co-analysis. On the other hand, an embedded encryption application (*tea*8) does use shift operations, demonstrating that different applications use different functionalities and may expose different amounts of DTS. Co-analysis reports a constraint for the shifter select bit in `inst_alu` for *rle* but not for *tea*8.

Figure 3.3 illustrates how co-analysis can automatically identify constraints for unexercised nets in `inst_alu` with a simplified example (a processor with only 4 operation types). As described above, a select bit in `inst_alu` only toggles during the execution of an embedded software application if the application contains an instruction that executes on the function unit selected by the bit. Performing co-analysis on a thresholding application (*tHold*) generates toggles in the adder and comparator select bits in `inst_alu` (colored blue in Figure 3.3), since *tHold* contains an `inc` (increment) instruction (which executes on the adder) and a `cmp` (compare) instruction. The code does not contain any `and` or `shift` instructions, however, so the corresponding select bits remain constant at zero during co-analysis of *tHold*. Applying these constraints propagates a controlling value to the select gates for the corresponding functional units and eliminates the logic (labeled inactive) from consideration during application-specific timing analysis, potentially exposing DTS.

3.4 Results

Evaluation of the methodology shows potential for significant power savings from exploiting DTS through application-specific timing analysis. Table 3.2 presents the power reduction for an openMSP430 processor afforded by DTS identification based on hardware-software co-analysis and application-specific timing analysis. The table shows the amount of DTS as a percentage of the clock period, the minimum safe operating voltage reported, and the power savings afforded for each application. The voltage reduction allowed from exploiting DTS is non-speculative and requires no reduction in operating frequency, so reported benefits are essentially "free" power savings. The baseline for the results is the processor operating at nominal frequency and voltage (100MHz and 1V for openMSP430). Over the range of applications, average power savings from exploiting DTS are 25%.

The results in Table 3.2 show that different applications can expose different amounts of DTS, resulting in different minimum safe operating voltages. This is because different applications exercise different processor features, resulting in a different set of logic constraints. For example, relatively less DTS is available for *AutoCorr*, *FFT*, *intFilt*, and *mult*, because these applications use openMSP430's hardware multiplier – one of

Table 3.2: Power savings from exploiting DTS.

Benchmark	DTS (%)	$V_{min}(V)$	Pwr Saved (%)
binSearch	32.01	0.86	28.38
div	31.42	0.86	28.42
inSort	31.79	0.86	28.40
intAVG	31.73	0.86	28.36
intFilt	21.64	0.90	21.10
mult	12.34	0.94	13.54
rle	31.76	0.86	28.34
tHold	32.13	0.86	28.37
tea8	31.42	0.86	28.36
AutoCorr	10.04	0.95	11.15
ConvEnc	31.73	0.86	28.39
FFT	10.04	0.95	11.14
Viterbi	31.43	0.86	28.50

the most timing-critical modules in the processor.

3.4.1 Comparison with Related Work

DVFS:

In this work, we exploit DTS for power reduction by reducing voltage without reducing frequency, such that all exercised parts of a processor design meet timing constraints. Related work on DVFS [40, 41, 42, 43, 44, 45] also reduces power by reducing voltage; however, DVFS reduces frequency along with voltage to ensure timing safety. Figure 3.4 compares power and energy reduction achieved by DVFS and DTS at different DVFS operating points (V/f). The "Power" series in Figure 3.4 shows power reduction for DVFS along with additional power reduction achieved by exploiting DTS at each operating point. DTS is orthogonal to DVFS and as such, DTS can be exploited for additional power savings at any DVFS operating point, even at the nominal operating point, where DVFS is not exploited to reduce power. Furthermore, while DVFS may lead to significant performance reduction, especially when performance is strongly correlated with frequency (e.g., in a system with embedded memories, like openMSP430), exploiting DTS at any operating point introduces no additional performance degradation, since DTS allows voltage reduction without any frequency reduction. Results for power-delay product (PDP) reduction show that DVFS can lead to an increase in energy (negative PDP reduction) at some operating points while DTS always reduces energy.

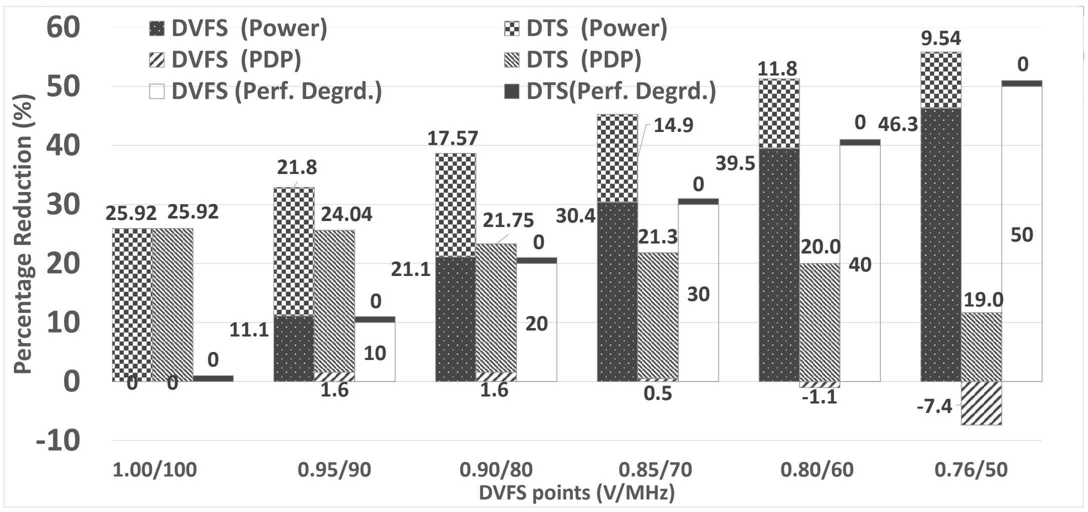


Figure 3.4: DVFS reduces frequency along with voltage and may lead to performance degradation. DTS enables voltage reduction without any reduction of frequency. Furthermore, the benefits of DTS are orthogonal to those of DVFS and can be extracted in addition to any benefits produced by DVFS.

Better-than-worst-case Design:

Our DTS identification methodology performs analysis at the worst-case design corner, leaving unexploited benefits at better-than-worst-case (BTWC) operating conditions. Below, we describe how our DTS identification approach can be combined with BTWC design techniques to reclaim benefits of guardband reduction while safely exploiting DTS. We also compare our approach for exploiting DTS against two popular BTWC design techniques – critical path monitors (CPMs) [46] and Razor [34, 35, 36].

CPMs: CPMs exploit static timing slack by monitoring circuits that track the static critical paths of a processor and adjusting the voltage to ensure that the circuit and processor meet timing constraints when the processor operates at an aggressive BTWC operating point. CPMs are less intrusive and have lower design and verification overhead than many comparable BTWC techniques and may also be more conservative, since they cannot track local process, voltage, and temperature (PVT) variations. For our evaluations of designs that employ CPMs, we select the operating point to maintain guardbands for local PVT variations. Compared to the power of the processor, the power overhead of CPM circuits is negligible.

The timing slack in guardbands under BTWC conditions (exploited by CPMs) is orthogonal to DTS (timing slack between un-exercised static critical paths and exercised dynamic critical paths). As such, DTS exploitation techniques can be used synergistically with CPMs for additional power reduction by using CPMs to track dynamic critical path delay rather than static critical path delay. We refer to CPMs that track dynamic critical path delay as dynamic critical path monitors (DCPMs) as opposed to conventional static critical path monitors (SCPMs). Since our DTS identification techniques identify the dynamic critical paths exercised by an application, tuning CPMs to track dynamic critical path delay is feasible using tunable CPMs [47, 48].

Razor: Razor introduces error detection and correction circuitry to a processor and adjusts the processor's voltage to operate at the minimum energy operating point, close to the point of first failure. Since Razor determines an aggressive operating voltage by observing when errors exceed a predefined threshold, it can eliminate guardbands and also exploit DTS. While Razor can potentially exploit DTS, it suffers from adding non-trivial area, design, and verification overheads making it unpalatable to ultra-low-power processors. Our approach, on the other hand, is non-speculative – software analysis determines an application-specific V_{min} (or f_{max}) that is guaranteed to be safe, irrespective of the input or operating conditions, since we perform input-independent analysis at the worst-case (slow) corner. As a result our technique has little or no hardware overhead and provides benefits even during worst-case operating conditions. Finally,our approach for exploiting DTS can even be used for existing processors and applications, without need for re-designing and re-certifying the processor.

To evaluate Razor, we first identify flip-flops (FFs) that need to be replaced with Razor FFs by selecting the minimum safe operating voltage for the processor under typical case operating conditions and identifying all the FFs that can violate timing constraints at this voltage under worst case operating conditions. After replacing these FFs with Razor FFs containing an extra (shadow) latch, clock buffer, XOR gate for error detection, and MUX for error correction, an "OR" network was added to combine the error signals to be sent to the voltage regulator, and hold time constraints were placed on the Razor FFs during layout of the synthesized netlist to generate the placed and routed netlist. Implementing Razor in this fashion resulted in an area overhead of 14% for openMSP430. Note, however, that this is an optimistic evaluation of Razor, as the Razor overheads for meta-stability detectors, error correction (dynamic performance and power overheads), clock gating, error rate measurement, and voltage control logic were not considered. Also, the design was not able to meet the hold time constraint for all Razor FFs (one of several difficult challenges for Razor designs [36]). Although we did not account any error correction overheads, we evaluated the Razor-based design at a reduced voltage corresponding to a 1% error rate for each benchmark.

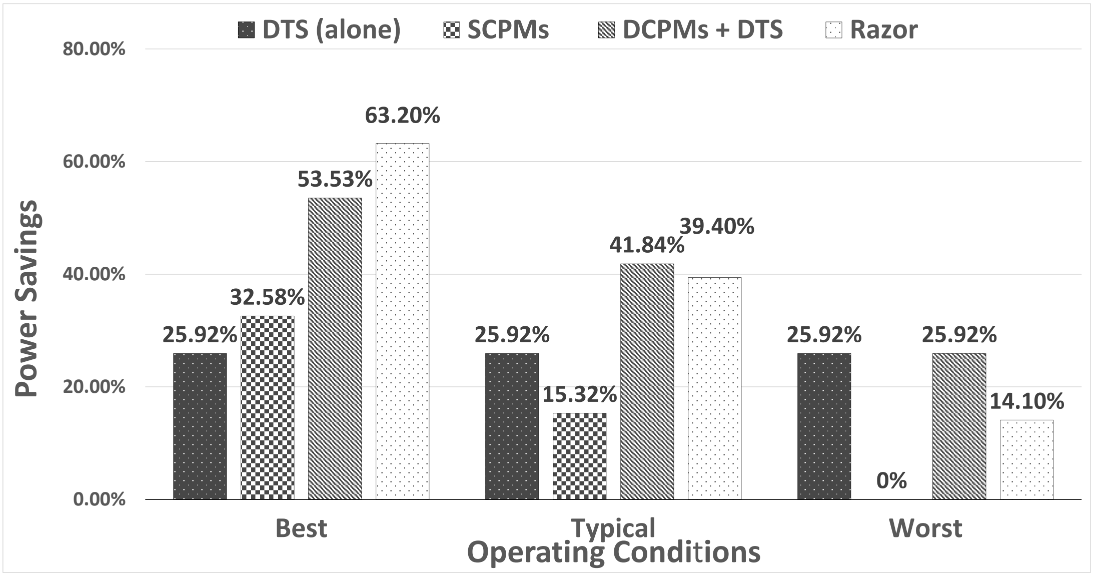


Figure 3.5: Comparison of power savings for DTS, Razor, and CPMs under different operating conditions.

Figure 3.5 compares power reduction achieved by DTS exploitation, SCPMs, DCPMs+DTS, and Razor under different operating conditions (worst, typical, best). SCPMs achieve significant power reduction at BTWC operating points (typical, best) but no reduction under worst case conditions. DTS, however, can be exploited for significant power savings (25%) even in worst case conditions. Exploiting DTS synergistically with CPMs (DCPMs+DTS) achieves significant additional benefits over SCPMs at BTWC operating points.

As mentioned above, Razor can potentially exploit DTS in addition to static timing slack resulting from BTWC operating conditions. Under best-case conditions, Razor can reduce power more than DTS+DCPMs, since CPMs maintain guardbands to protect against local variations. Under worst-case conditions, exploiting DTS (with or without DCPMs) reduces power more than Razor, even though Razor exploits DTS. This is due to the power overheads associated with Razor-based design. Nevertheless, both best-case and worst-case conditions are rare. Under typical conditions, Razor and DCPMs+DTS achieve similar power savings. However, our automated techniques for exploiting DTS may be more attractive, especially in ultra-low-power embedded designs, due to the area, design, and verification overheads of Razor.

Chapter 4

Application-specific Power Gating

The activity profile generated by hardware-software co-analysis (Chapter 2) can be used to identify times when an application is guaranteed not to exercise a specific set of gates in the processor. Information about when a set of gates is guaranteed to be inactive can be produced at any granularity, ranging from a single gate to the entire processor. This information can be used to group gates into application-specific power domains based on correlated periods of inactivity and to inform the processor when to power a set of gates (domain) on or off to save power for a specific application.

While a large body of work exists on power gating at the processor or core level [55, 56, 57, 58, 59, 60], emerging power- and energy-constrained applications have fueled recent work on aggressive module-based power gating techniques, in which register-transfer level (RTL) modules are powered down during periods of inactivity [61, 59, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72]. Table 4.1 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 4.1: Power Domains in Recent Processors

Processor	#domains
TI MSP430 Wolverine	"many" [49]
ARM Cortex-A9	14 [50]
ARM Cortex-A15	8 [51]
Atmel SAML21	5 [52]
Intel Atom E6	15+ [53, 54]

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 and not necessarily correlated activity. In this section, we propose research on application-specific *module-oblivious* power gating. A module-oblivious power domain is an arbitrary set of gates that have correlated activity profiles, as determined by hardware-software co-analysis (Chapter 2). Module-oblivious power domains may contain only a subset of gates in a module, may contain gates from multiple (often many) 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**.

4.1 A Case for Module-oblivious Power Domains

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, it is common for several modules to contain some 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. Similarly, logic that spans across module boundaries may exhibit correlated activity, such as logic in one module that drives logic in another module. Although the entire modules containing the driving and driven logic 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 4.1 shows activity profiles for the frontend and execution unit modules of the openMSP430 processor [26], 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 execution unit have at least one part active during nearly every instant of

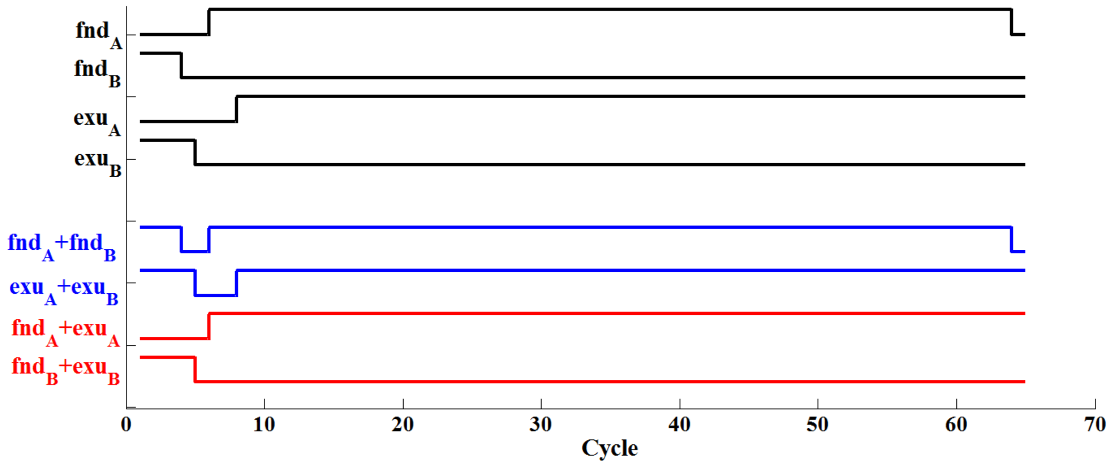


Figure 4.1: Uncorrelated activity within a module can prevent power gating of module-based domains, whereas module-oblivious domains allow more aggressive power gating.

this time period, there is no opportunity to power gate either module (see blue activity profiles). 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 idle. 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 (see red activity profiles). Uncorrelated activity within modules and correlated activity across module boundaries indicates that there may be significant opportunities to perform more aggressive power gating with module-oblivious power domains.

Figure 4.2 is a heat map that shows the correlation between each pair of sub-modules in the openMSP430 processor, where correlation equals the fraction of cycles in which two sub-modules exhibit the same activity (active or idle). The dashed black 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

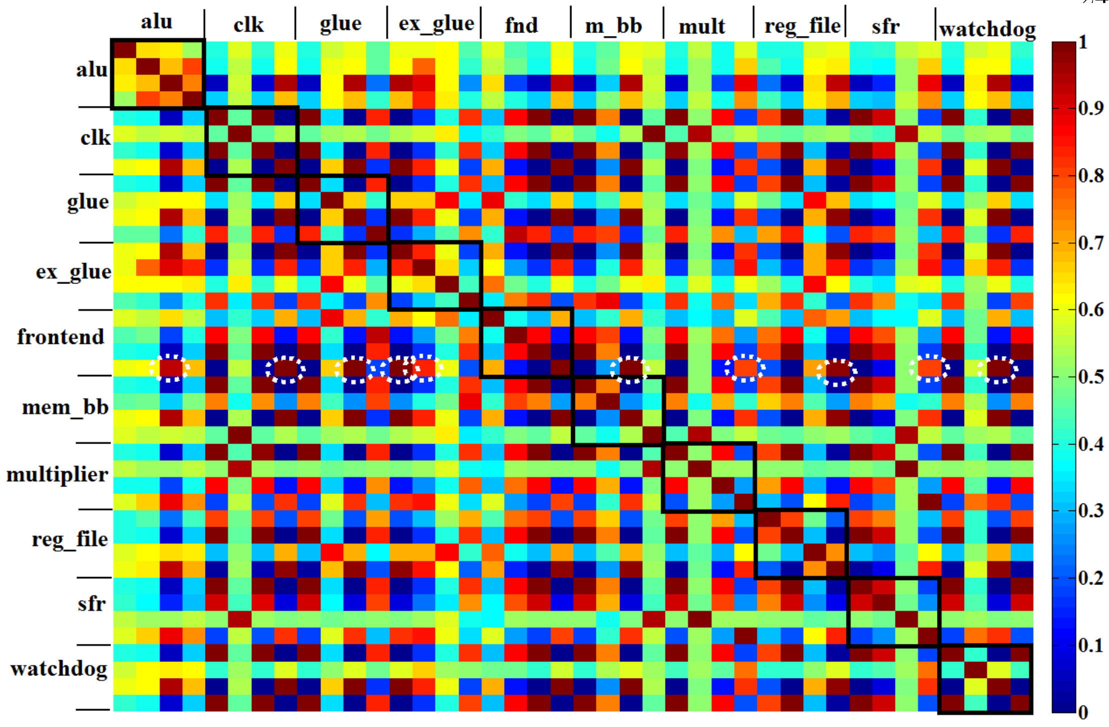


Figure 4.2: Different parts of the same module can have uncorrelated activity profiles. For nearly all module partitions, there is a partition from a different module with more correlated activity than another partition from the same module.

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.

4.2 Forming Module-oblivious Power Domains

One approach for module-oblivious domain formation is to use hierarchical agglomerative clustering [73], in which every *gate* is initially placed in a separate cluster, and clusters are combined until the desired number of domains is formed. 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 hardware-software co-analysis of representative applications) to determine which

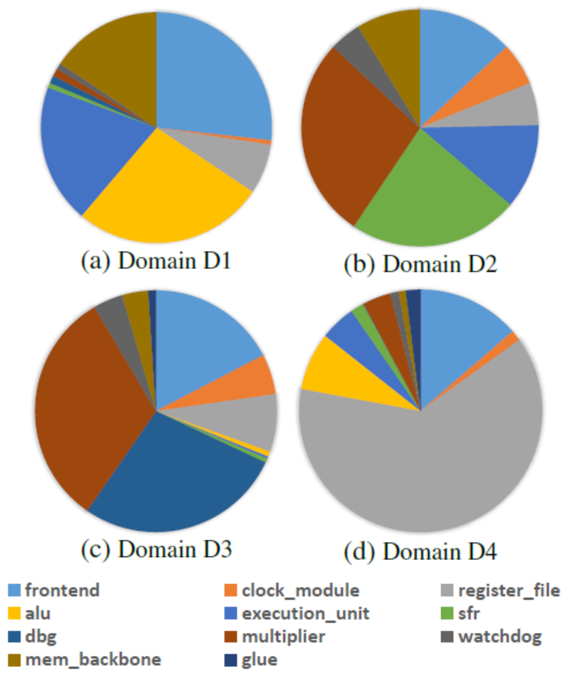
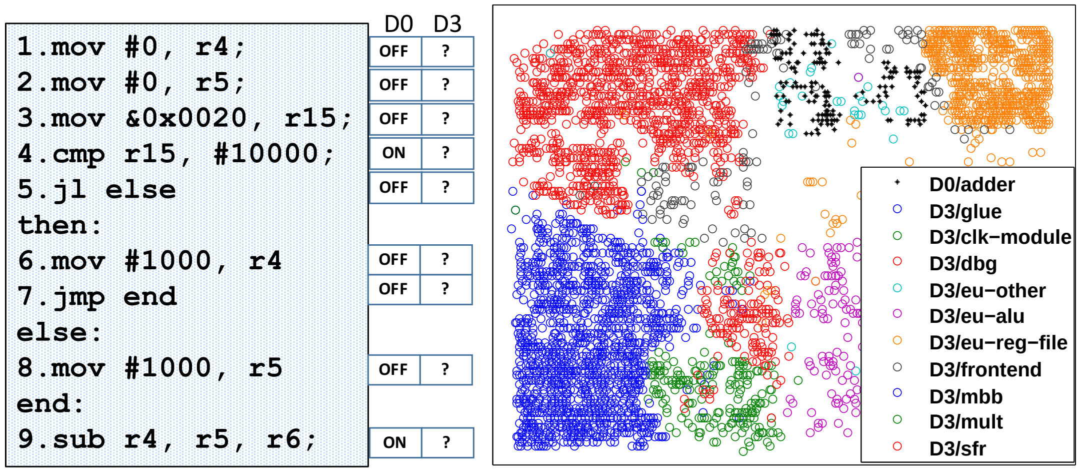


Figure 4.3: Breakdown of domain composition for four module-oblivious power domains formed using hierarchical agglomerative clustering. All four domains have gates from at least eight microarchitectural modules.

combination of clusters maximizes the potential energy savings achieved by power gating the resulting domains. Since a gate may end up in a domain containing gates from other modules, the resulting domains are module-oblivious. Figure 4.3 shows the composition of four module-oblivious power domains created using the approach described above.

4.2.1 Application-specific Power Domain Management

To reap the potential power benefits provided by module-oblivious domains, a power domain management technique is needed that can determine when domains are idle / active and power them off / on accordingly. Unfortunately, existing techniques that are used to manage module-based domains through software or hardware cannot be used for module-oblivious domains. Consider existing software-based management techniques [74, 61]. These techniques infer when power domains will be inactive by analyzing an application binary. Software-based management is possible when the activity of domains can be inferred from software, as is the case for many module-based domains [55]. In the example code listing in Figure 4.4, 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 from software when the domain needs to 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.



```
                          D0   D3
1.mov #0, r4;             OFF  ?
2.mov #0, r5;             OFF  ?
3.mov &0x0020, r15;       OFF  ?
4.cmp r15, #10000;        ON   ?
5.jl else                 OFF  ?
then:
6.mov #1000, r4           OFF  ?
7.jmp end                 OFF  ?
else:
8.mov #1000, r5           OFF  ?
end:
9.sub r4, r5, r6;         ON   ?
```

Figure 4.4: 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) based on software alone.

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, making it impossible to infer activity of module-oblivious domains from software alone. For example, domain D3 in Figure 4.4 corresponds to a module-oblivious domain containing gates from ten different modules. The collection of gates has no architectural meaning, and it is not possible to infer when the gates will be active or inactive based on software alone.

```
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
```

Figure 4.5: Verilog statements for inferring the activity of the execution unit adder module synthesize to 6 gates.

Similarly, existing hardware-based domain management techniques are not feasible for module-oblivious domains. Hardware-based domain management techniques use hardware monitors to dynamically determine when a power domain is idle / active based on processor control signals [57, 56, 58, 72, 66, 67]. 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 4.4 – 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 4.5 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 module-oblivious domains 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. As such, the number of signals that must be monitored to infer domain activity is prohibitively large. Figure 4.6 shows the verilog statements that infer the activity of the module-oblivious domain D3 from Figure 4.4. Synthesized, this logic corresponds

```
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
```

Figure 4.6: Verilog statements for inferring the activity of the module-oblivious domain D3 from Figure 4.4 synthesize to 4010 gates.

to 4010 gates. For the entire openMSP430 processor, we estimated the area and static power overheads of hardware-based management of four module-oblivious domains to be 184% and 174%, respectively, precluding any possible power reduction from aggressive power gating.

Any viable technique for managing module-oblivious power domains must be based on inferring software-induced activity at the gate level such that (a) the activity of module-oblivious domains can be inferred and (b) the prohibitive overheads associated with hardware-based monitoring of an arbitrary set of gates can be avoided. The annotated symbolic execution tree produced by hardware-software co-analysis can be used, along with gate-to-domain mapping information, to annotate static instructions in a binary with power gating decisions for all domains in the processor. All domains that contain at least one active gate in a given cycle must be turned on. To ensure that a domain is turned on in time, binary annotation 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 on the domain.

If a domain is not active for any dynamic instance of a particular static instruction (even considering wakeup overheads), the domain can be powered off. The annotated binary contains application-specific domain management decisions, indicating when each domain must be turned on and when each domain can be turned off. We will explore a variety of microarchitectural support mechanisms for communicating software-based power gating decisions to domain control logic. Since the decisions are annotated in the application binary, microarchitecture support mechanisms used in previous works on software-based power gating [74] can be used to communicate power gating decisions determined by co-analysis, including insertion of power gating instructions, utilizing reserved instruction bits, and storing gating decisions in a table referenced by an instruction's address (PC).

We now illustrate the proposed approach for management of module-oblivious power domains with an example. Figure 4.7 revisits the example code from Figure 4.4 to demonstrate that the proposed technique based on hardware-software co-analysis *can* infer the activity of module-oblivious domains, which was impossible to infer from software alone.

Figure 4.7 shows the annotated symbolic execution tree generated by co-analysis. Co-analysis 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 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, each dynamic instruction is annotated with domain activity for each domain (D1 and D2 in Figure 4.7). 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 during that instruction. Next, the domain states (ON/OFF) from the symbolic execution tree are mapped to the static instructions in the application

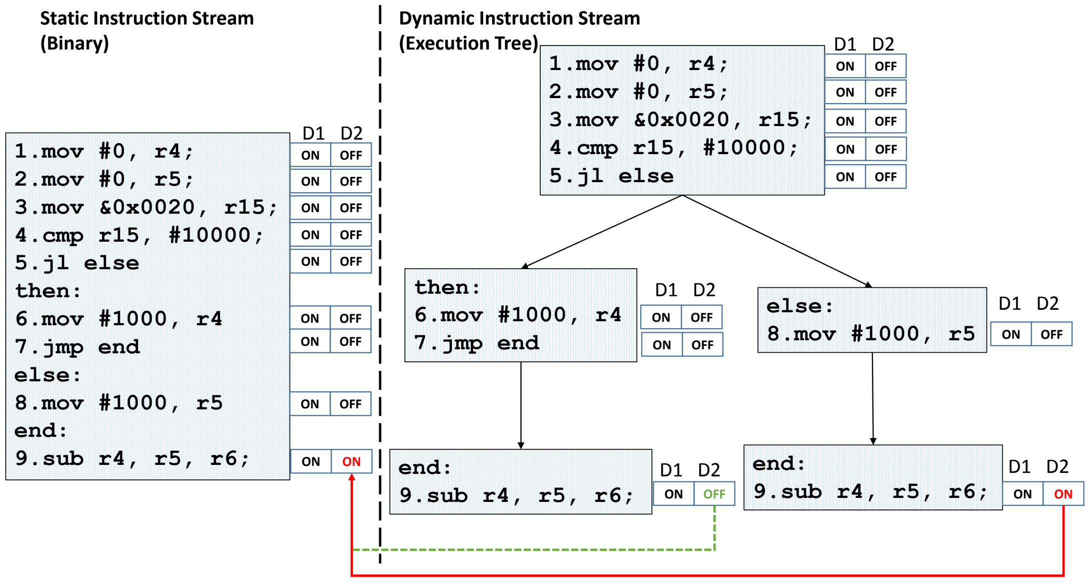


Figure 4.7: Illustration of binary annotation for application-specific power gating of an example code (an if-else block). For simplicity, this example only shows domain-level activity, assumes that each instruction takes a single cycle, and assumes a wake-up latency of zero cycles.

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, the corresponding static instruction is annotated 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 execution tree. The activity of D1 is consistent across the two instances (D1 is ON for both); therefore, the static instruction is annotated 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, the conflict is resolved conservatively by marking D2 as ON in the static instruction annotation. This ensures safety for all possible executions of the application.

4.3 Microarchitecture Support for Software-based Power Gating

The previous section describes 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 4.7 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 [74].

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.

4.4 Results

Evaluation of the methodology suggests that application-specific power gating of module-oblivious domains can have significant benefits over state-of-the-art module-based power gating. Figure 4.8 compares the leakage energy savings provided by our approach (co-analysis) against (1) state-of-the-art hardware-based management based on Idle-Count [75] with different idle periods (5, 10, 100) and (2) oracular management, which shows how our approach compares to optimal power gating. Results are shown for four power domains and different domain wakeup latencies.[1] 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 domain isolation and state retention overheads 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 unified power format (UPF) to fully implement power gating designs and accurately account for all

[1] The domain wakeup latency of 1 cycle is the most realistic for most small embedded processors [76, 58] We also evaluated 100- and 1000-cycle wakeup latencies, which show the same trend as 10-cycle results. Additionally, evaluations for two and three power domains show the same trend.

implementation overheads of power gating. Note also that for our instrumentation overheads, these preliminary results conservatively assume the software approach with the highest overhead – binary instrumentation with dedicated power gating instructions.

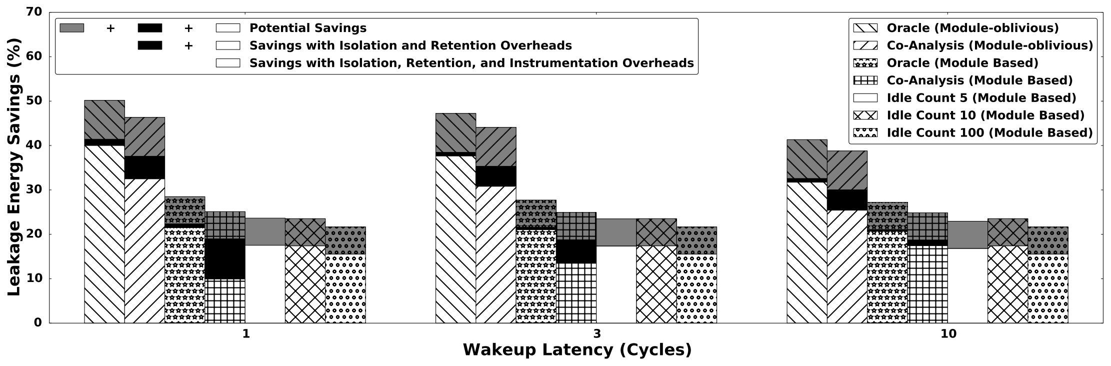


Figure 4.8: Comparison of leakage energy savings provided by different domain management and formation techniques. Results in each stack (from top to bottom) correspond to maximum potential benefits of the technique, benefits after accounting for isolation and retention, and benefits after isolation, retention, and instrumentation.

On average, power gating on module-oblivious domains provides 2× more savings than hardware-based management of module-based domains. Preliminary results show that co-analysis is a very effective management technique for module-oblivious domains, as its power benefits are within 3.0% of optimal (oracle) management of module-oblivious domains. Co-analysis can also be used to generate domain management decisions for existing module-based domains, saving 6.0% *more* energy than hardware-based domain management for module-based domains, even when assuming no hardware overhead for implementing IdleCount.

A preliminary analysis of the performance impact of instrumentation overhead for module-oblivious power gating shows that application-specific power gating based on co-analysis has lower overhead than state-of-the-art hardware-based domain management (IdleCount), even assuming the most costly proposed binary instrumentation technique (dedicated power gating instructions). Note that existing software-based power gating approaches would have comparable instrumentation overhead to power gating based on co-analysis; however, existing software-based approaches cannot manage module-oblivious domains, and thus, cannot tap into the higher level of leakage savings that they provide over module-based domains.

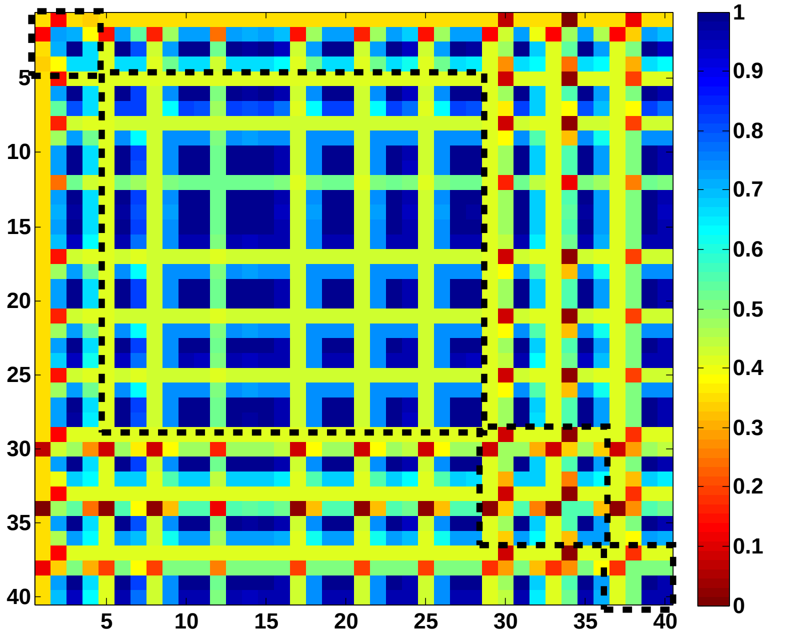


Figure 4.9: "Cool" map for module-based domains shows little opportunity for saving power in the processor

Figure 4.9 and Figure 4.10 are visualizations of opportunites to save power using power gating for module-based and module-oblivious power domains, respectively. Contrasting these figures shows why module-oblivious domains provide more opportunities for power gating than module-based domains. Each 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, where power gating correlation is defined as the fraction of cycles that the two sub-modules, sub-module$_1$ and sub-module$_2$, are 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 4.9 shows power gating correlation for module-based domains, and Figure 4.10 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.

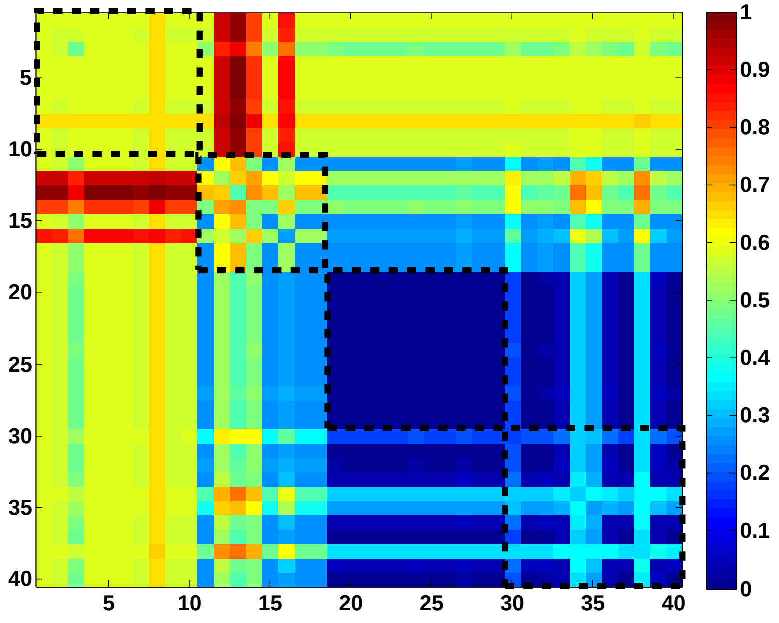


Figure 4.10: "Cool" map for module-oblivious domains shows a large opportunity for saving power in the processor

Module-based domains do not account for the fact that different parts of the same microarchitectural module 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 4.9 are "cool", the small number of "hot" cells in each domain prevent many power gating opportunities. Figure 4.9 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 4.8 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.

Chapter 5

Application-specific Peak Power and Energy Estimation

Since co-analysis of hardware and software (see Chapter 2) can identify all instances when the gates in a processor can possibly toggle for an application, the results of co-analysis can be used to determine application-specific peak power and energy requirements for a low-power embedded system. Low-power systems can be classified into three types based on how they are powered [77]. As illustrated in Figure 5.1, some are powered directly by energy harvesting (Type 1), some are battery-powered (Type 3), and some are powered by a battery and use energy harvesting to charge the battery (Type 2). For each of these classes, the size of energy harvesting and/or storage components determine the form factor. Consider, for example, the wireless sensor node in Figure 5.2 [78]. The two largest system components that predominantly determine the 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 low-power system are determined by its power and energy requirements, the peak power and energy requirements of a low-power system are the primary factors that determine critical system characteristics such as size, weight, cost, and lifetime [77]. 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

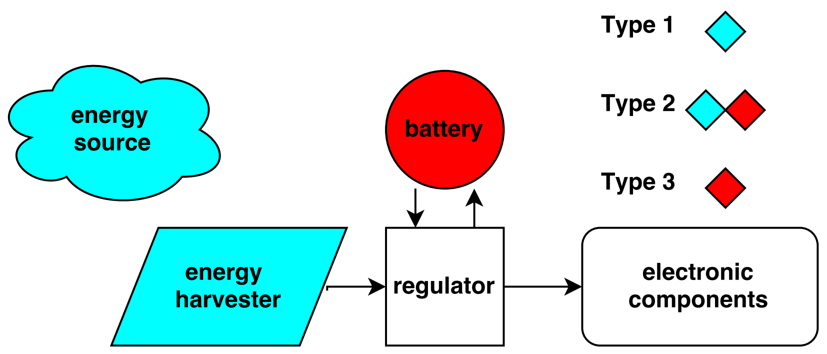


Figure 5.1: Low-power systems are commonly powered by energy harvesting, battery, or a combination of the two, where harvesters are used to charge the battery.

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* [12]. As battery discharge rate increases, effective capacity drops [13, 12]. This effect is exaggerated in low-power systems, where near-peak power is consumed for a short 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[13].

In Type 2 and 3 systems, peak energy requirements also matter. 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) [13]. Figure 5.3 summarizes how peak power and energy requirements impact sizing parameters for the different classes of low-power systems. Finally, Tables 5.1 and 5.2 list the power and energy densities for different types of energy harvesters and batteries, respectively. These data provide a rough sense of how size and weight of a low-power system scale based on peak power and energy requirements.

Table 5.1: Power density for different types of energy harvesters. [2]

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

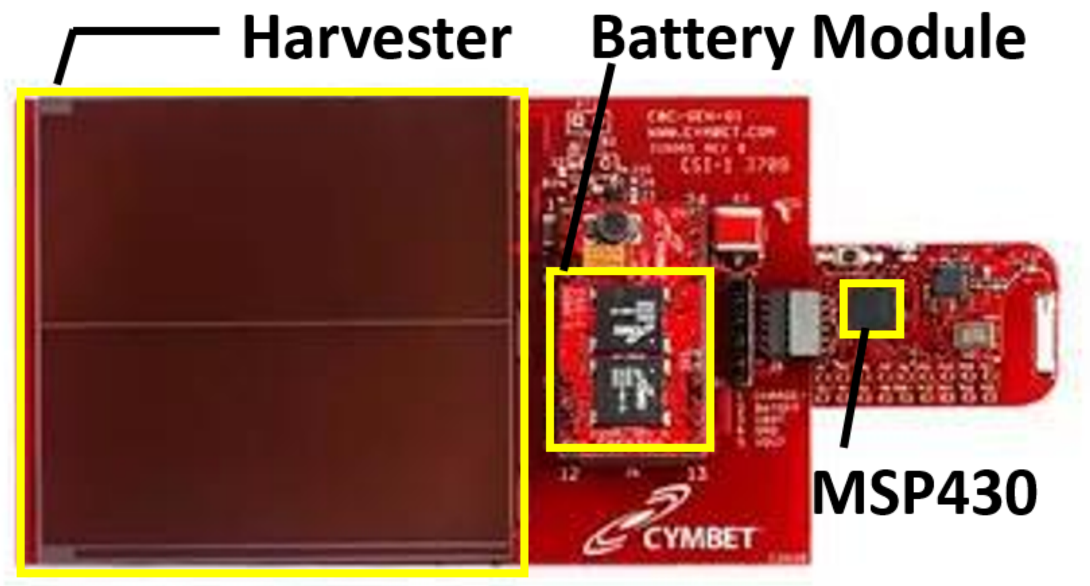


Figure 5.2: In most low-power systems, like this wireless sensor node, the size of the battery and/or energy harvester dominates the total system size.

5.1 Conventional Peak Power and Energy Estimation

Table 5.2: Specific energy and energy density for different battery types [3].

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

There are several possible approaches to determine peak power and energy requirements for a low-power processor (Figure 5.4). The most conservative approach involves using 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 not application-specific and is thus conservative; however, it is safe for any application that can 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

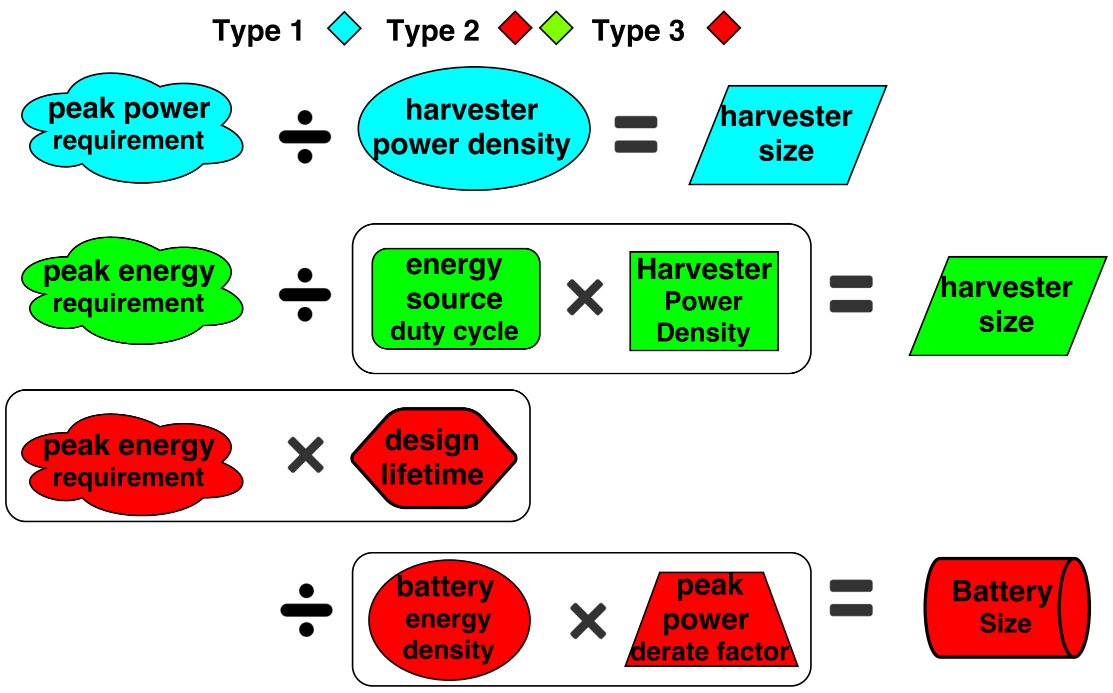


Figure 5.3: Harvester and battery size calculations for Type 1, 2, and 3 low-power systems.

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 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 low-power processor is to perform application profiling on the processor by measuring power consumption while running the target application on the hardware. 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.

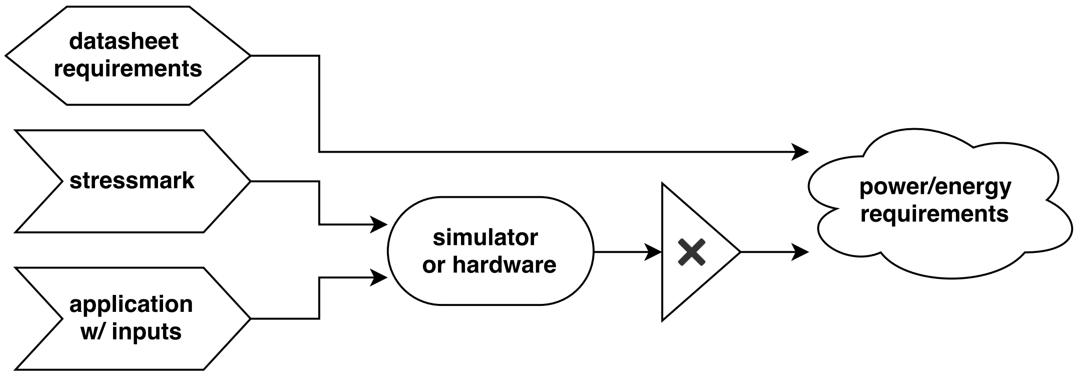


Figure 5.4: The conventional methodology for sizing energy harvesting and storage components involves determining peak power and energy requirements for a processor and selecting components that provide enough power and energy to satisfy the requirements over the lifetime of the system.

5.2 Input-Independent Peak Power Requirements

Most low-power processors run the same application or computation over and over in a compute / sleep cycle for the entire lifetime of the system [79]. As such, the power and energy requirements of low-power 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.

However, while the peak power and energy requirements of low-power processors tend to be application-specific, conventional design-based and stressmark-based techniques for determining peak power and energy requirements are not application-specific. A profiling-based approach is application-specific, but it must use guardbands to inflate the peak power requirements observed during profiling, since it is not possible to determine bounds that are guaranteed for all possible input sets. Existing conventional techniques cannot accurately bound the power and energy requirements of an application running on a processor, leading to over-provisioning that increases critical system characteristics like size, weight, and cost.

Motivation for input-independent application-specific peak power and energy requirements

Preliminary measurement-based results provide motivation for application-specific peak power and energy requirements. Figure 5.5 compares the peak power observed for different applications running on an MSP430F1610 processor.[1] The results show that peak power can be different for different applications. Thus, a peak power bound that is not application-specific will overestimate the peak power requirements of applications, leading to over-provisioning of energy harvesting and storage components that determine system size and weight. Figure 5.5 also shows that the peak power requirements

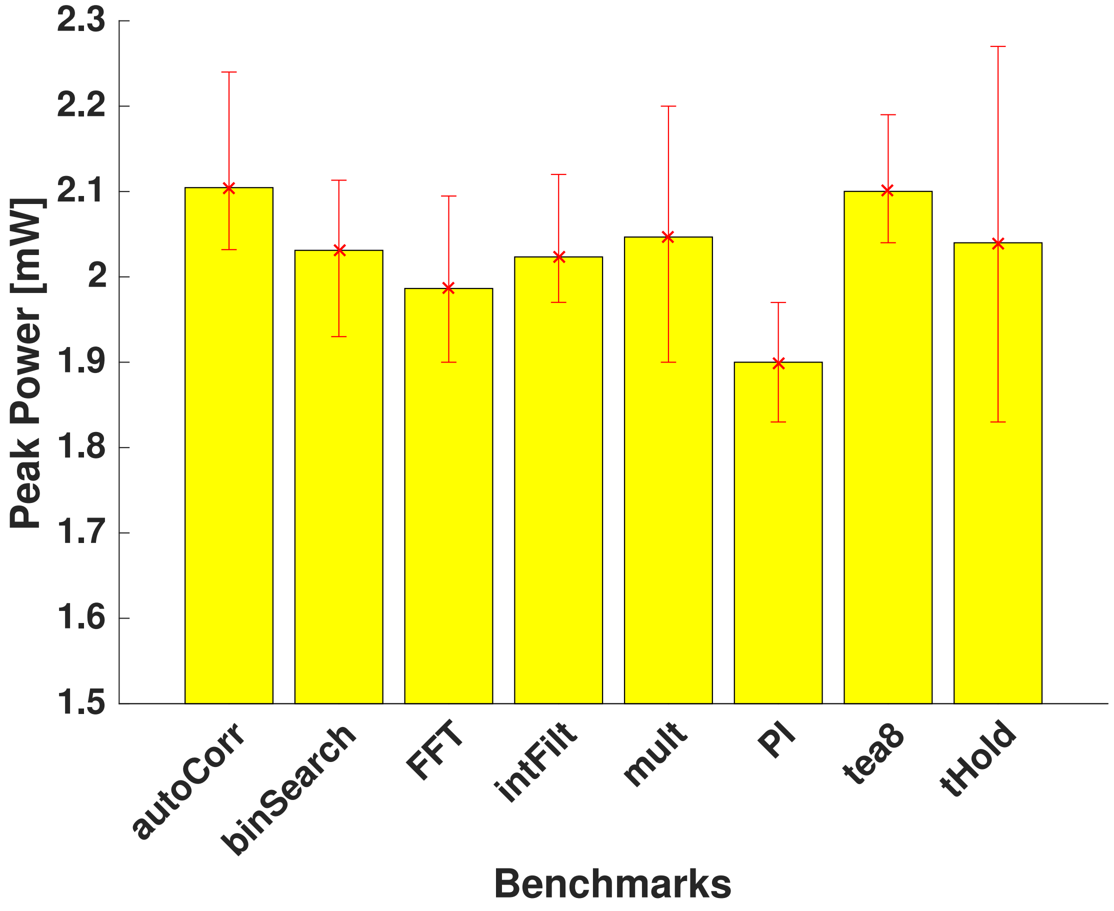


Figure 5.5: The peak power of a low-power processor is different for different applications and different inputs. Input-induced peak power variations are shown as error bars.

of applications are significantly lower than the rated peak power of the chip (4.8 mW), so using design specifications to determine peak power requirements can lead to significant over-provisioning and inefficiency. The figure also confirms that peak power of an

[1] MSP430 is one of the most popular processors used in low-power systems [49, 80]

application depends on application inputs and can vary significantly for different inputs (input-induced variations shown as error bars). For different inputs, the applications in Figure 5.5 show input-induced variations in peak power of over 25%. This means that profiling cannot be used to accurately determine peak power requirements, 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.

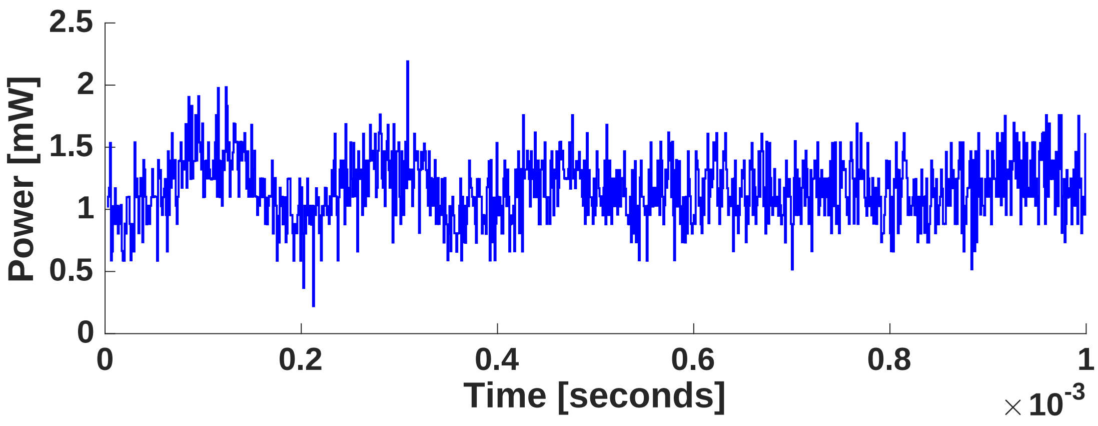


Figure 5.6: Measured instantaneous power of MSP430F1610 for an application is significantly lower, on average, than both the rated and observed peak power.

In energy-constrained systems, like battery-powered systems (Type 2 and 3), both peak energy and peak power determine the size of energy harvesting and storage components. Thus, determining an accurate bound on the peak energy requirements of a low-power processor is also important. Figure 5.6 shows the instantaneous power profile for an embedded application, demonstrating that on average, instantaneous power can be significantly lower than peak power. Therefore, we can more accurately determine optimal component sizes in an energy-constrained system by generating an accurate bound on peak energy, rather than conservatively integrating rated power over time.

5.3 Determining Application-specific Peak Power and Energy

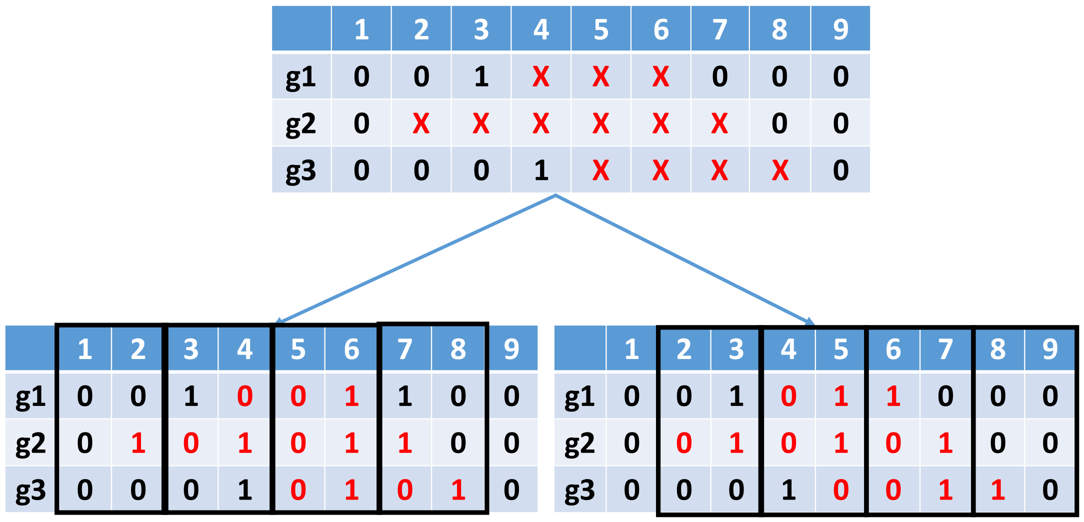
This section describes how the activity-annotated symbolic execution tree created by co-analysis (see Chapter 2) can be used to determine the peak power requirements of a low-power processor and application pair. Algorithm 2 describes how to use activity-annotated execution tree generated from hardware-software co-analysis (see Chapter 2) to compute peak power requirements for a ULP processor-application pair.

Algorithm 2 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($P_{peak}$)
```

The first step in determining the peak power from the symbolic execution tree is to concatenate the execution paths in the execution tree to form a single execution trace. The execution trace contains Xs, and the goal of the peak power computation is to assign values to the Xs in the way that maximizes power for each cycle in the execution trace. The power of a gate in a particular cycle is maximized when the output value of a gate transitions (toggles). Since a transition occurs over two cycles, maximizing the dynamic power in a particular cycle, c, of the execution trace involves assigning values to any Xs in the activity profiles of the current cycle, c, and the previous cycle, $c - 1$, 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 the values are Xs, the values are assigned to produce the transition that maximizes power in cycle c. The maximum power transition can be found by a look-up in 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 activity traces – 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 activity traces 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. A peak energy trace can be derived from the peak power trace by multiplying the per-cycle peak power values in the peak power trace by the clock period.



	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

Figure 5.7: To determine a bound on peak power, we generate two activity traces – one that maximizes power in even cycles (left) and one that maximizes power in odd cycles (right).

Figure 5.7 illustrates activity trace generation. We use the example of three gates with overlapping Xs that must be assigned to maximize power in every cycle. We show two assignments – one that maximize 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 $0 \rightarrow 1$ transition consumes more power than the $1 \rightarrow 0$ 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.

5.4 Results

Evaluation of the peak power and energy estimation methdologies suggest that application-specific peak power and energy bounds may enable significant benefits for low-power systems. Figure 5.8 compares peak power requirements reported by input-independent application-specific analysis (X-based) against the conventional techniques for determining peak power requirements (design specification, stressmark, and guardbanded input-based profiling). We also compare our input-independent approach against (input-based) profiling without guardbanding to evaluate how tight of a bound our technique provides. Our results show that the proposed technique provides the most accurate bound on peak power. Across a range of applications, the peak power requirements reported by the proposed technique are only 1% higher, on average, than the highest observed input-based peak power for the applications. 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. The peak power requirements reported by X-based application-specific analysis are 23% lower than guardbanded application-specific requirements, 30% lower than guardbanded stressmark-based requirements, and 27% lower than design specification-based requirements, on average.

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 X-based analysis becomes more conservative when there is greater possibility for input-dependent variation in power, because it provides a bound on power for all possible inputs. 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

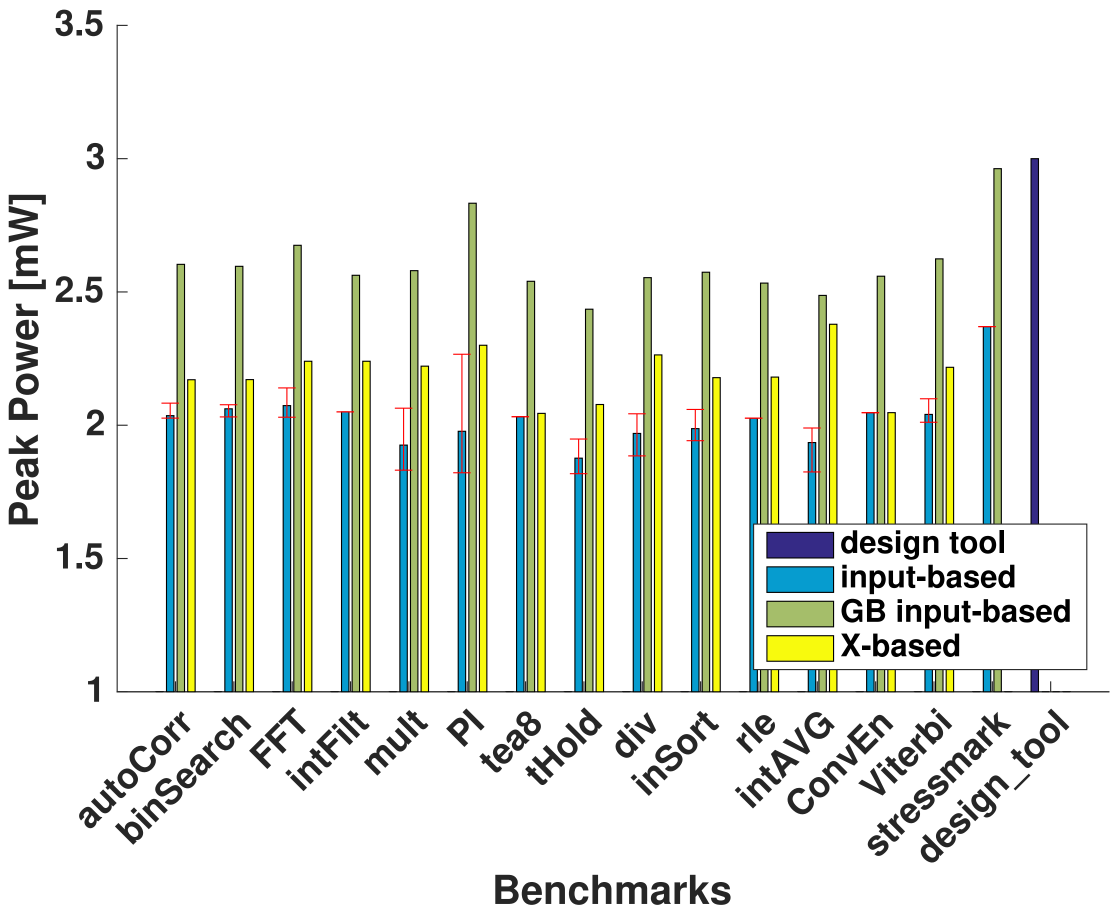


Figure 5.8: The proposed application-specific technique for determining peak power requirements provides the most accurate (least conservative) guaranteed bound on peak power.

highest possible power for a multiply 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, only uses 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 application, 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.

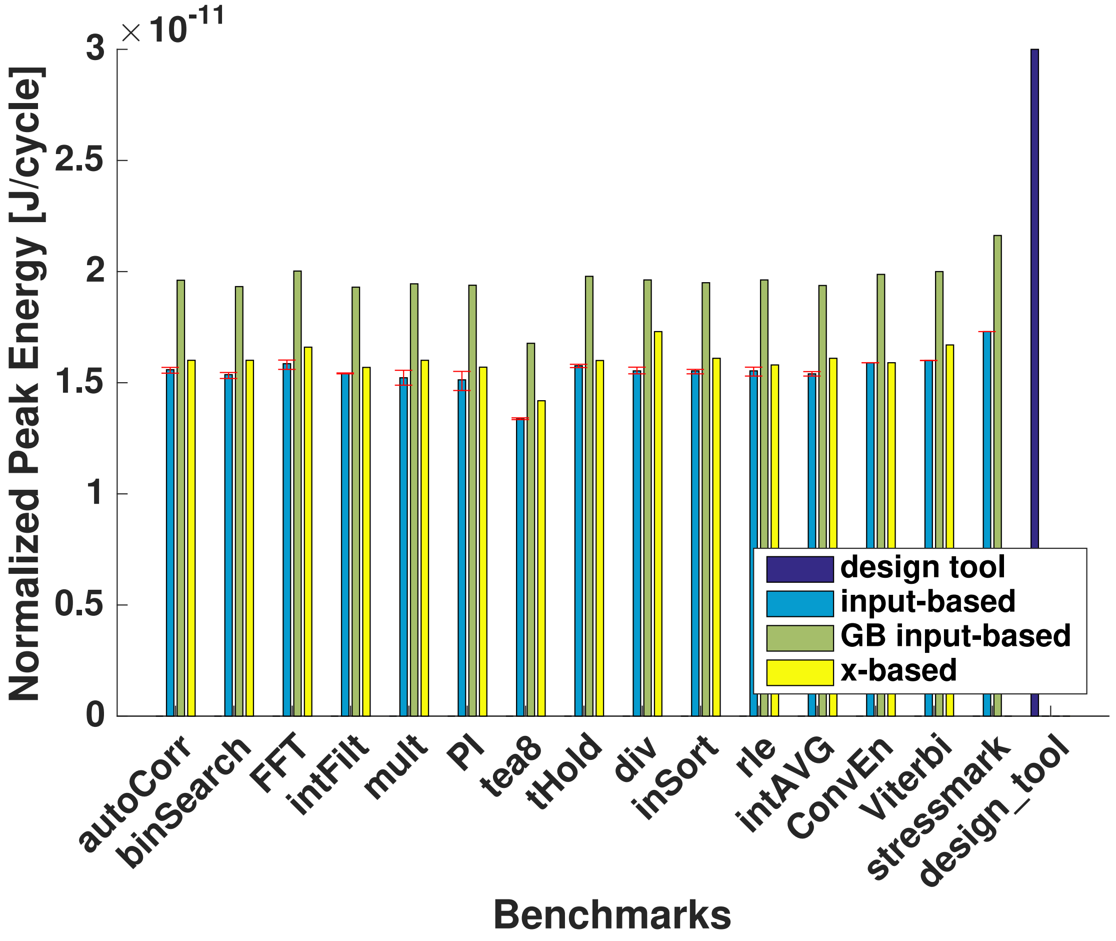


Figure 5.9: The proposed application-specific technique for determining peak energy requirements is more accurate than existing conventional techniques.

The proposed application-specific technique also shows potential to provide more accurate bounds on peak energy than conventional techniques. In Figure 5.9, the application-specific (X-based) peak energy requirements are 23% lower than guard-banded application-specific requirements, 31% lower than guardbanded stressmark-based requirements, and 48% lower than design specification-based requirements, on average. As described above, more accurate peak power and energy requirements can be leveraged to reduce critical system parameters like size and weight. For example, reduction in a Type 1 system's peak power requirements allows a proportionally smaller energy harvester to be used, and 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 3 systems.

5.4.1 Optimizations

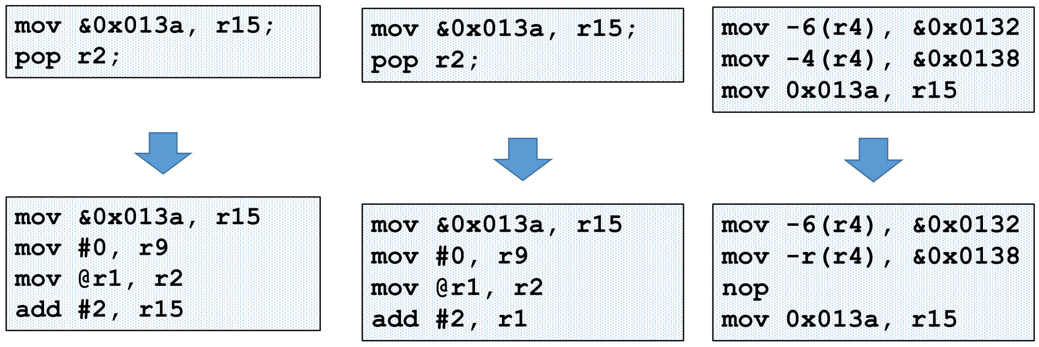
Our technique can be used to guide application-level optimizations that reduce peak power. Here, we discuss three software optimizations, based on by our technique, that we applied to the benchmark applications to reduce peak power. The optimizations were derived by analyzing the processor's behavior 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 since a `POP` instruction generates high activity on the data and address buses and simultaneously uses 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 5.10.



```
mov &0x013a, r15;
pop r2;
```

```
mov &0x013a, r15
mov #0, r9
mov @r1, r2
add #2, r15
```

(a) OPT 1

```
mov &0x013a, r15;
pop r2;
```

```
mov &0x013a, r15
mov #0, r9
mov @r1, r2
add #2, r1
```

(b) OPT 2

```
mov -6(r4), &0x0132
mov -4(r4), &0x0138
mov 0x013a, r15
```

```
mov -6(r4), &0x0132
mov -r(r4), &0x0138
nop
mov 0x013a, r15
```

(c) OPT 3

Figure 5.10: Instruction optimization transforms for peak power reduction.

• **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 where the value of the register is directly used as the memory address instead of as an offset. Using another instruction (such 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 `MOV`ed to the inputs of the multiplier and then the output is `MOV`ed 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.

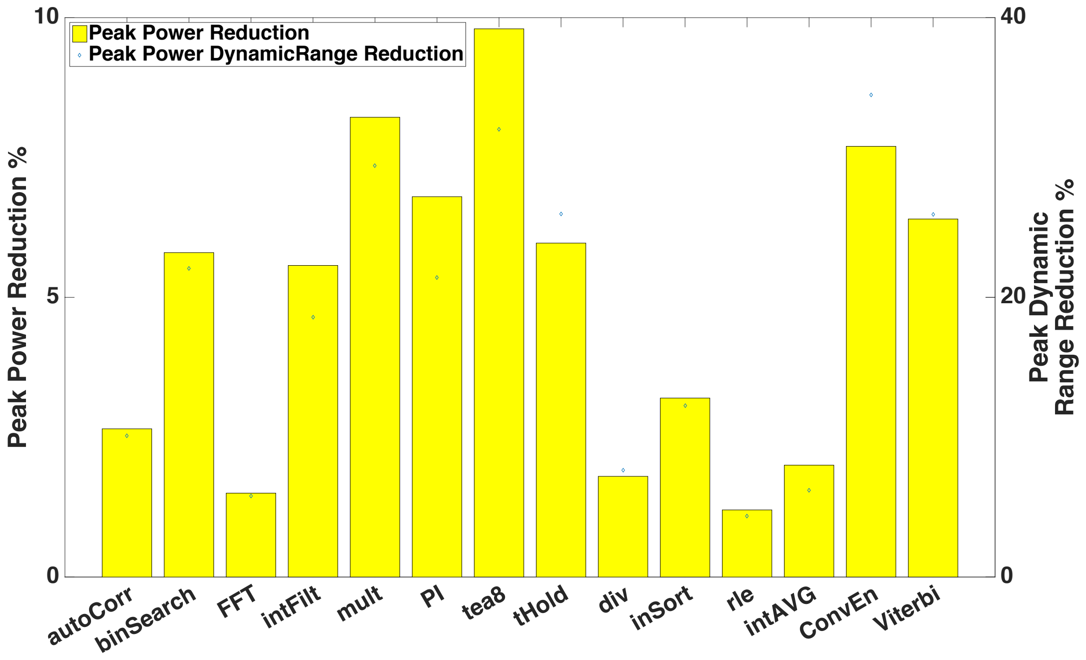


Figure 5.11: Peak power reduction (left axis) and peak power dynamic range reduction (right axis) achieved by optimizations. These reductions are enabled by our analysis tool and provide further reduction in energy harvester size.

Figure 5.11 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 3 systems, and reduction in peak power requirements can be leveraged to reduce harvester size in Type 1 systems. 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 5.12 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

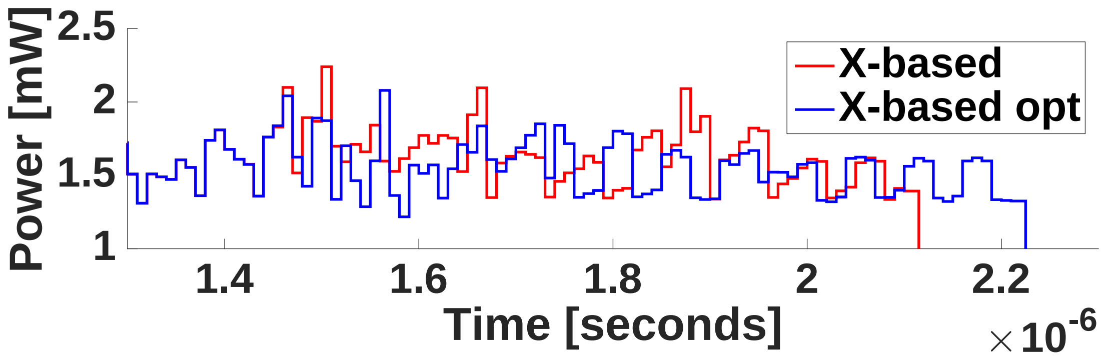


Figure 5.12: A snapshot of instantaneous power profiles for mult before and after optimization.

optimizations. Figure 5.13 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 3 systems, where both peak power and energy determine the size of energy storage and harvesting components (see Figure 5.3).

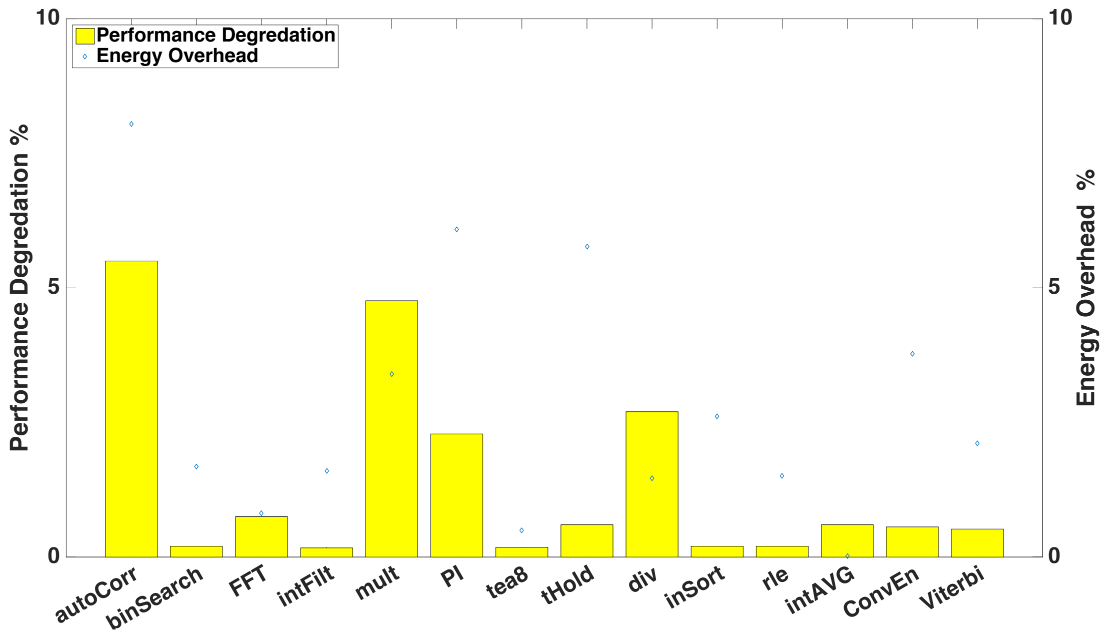


Figure 5.13: Performance degradation and energy overhead introduced by peak power optimizations is small(average: 1%).

Chapter 6

Application-specific ‘Bespoke’ Processors

One straightforward application of gate activity information gathered from hardware-software co-analysis in Chapter 2 is eliminating all the gates that are guaranteed not to be exercised for an application. A large class of emerging applications is characterized by severe area and power constraints. For example, wearables [5, 6] and implantables [81, 82] are extremely area- and power-constrained. Several IoT applications, such as stick-on electronic labels [83], RFIDs [84], and sensors [8, 85], are also extremely area- and power-constrained. Area constraints are expected to be severe also for printed plastic [86] and organic [87] applications.

Cost concerns drive many of the above applications to use general purpose microprocessors and microcontrollers instead of much more area- and power-efficient ASICs, since, among other benefits, development cost of microprocessor IP cores can be amortized by the IP core licensor over a large number of chip makers and licensees. In fact, ultra-low-area- and power-constrained microprocessors and microcontrollers powering these applications are already the most widely used type of processing hardware in terms of production and usage [9, 10, 11], in spite of their well-known inefficiency compared to ASIC and FPGA-based solutions [88]. Given this mismatch between the extreme area and power constraints of emerging applications and the relative inefficiency of general purpose microprocessors and microcontrollers compared to their ASIC counterparts, there exists a considerable opportunity to make microprocessor-based solutions for these applications much more area- and power-efficient.

One big source of area inefficiency in a microprocessor is that a general purpose microprocessor is designed to target an arbitrary application and thus contains many more gates than what a specific application needs (Figure 6.1). Also, these unused gates continue to consume power, resulting in significant power inefficiency. While adaptive power management techniques (e.g., power gating) help to reduce power consumed by unused gates, the effectiveness of such techniques is limited due to the coarse granularity at which they must be applied, as well as significant implementation overheads such as domain isolation and state retention. These techniques also worsen area inefficiency.

We propose a novel approach - *bespoke processors* - to significantly increase the area and power efficiency of a microprocessor for a given application by eliminating all logic in the microprocessor IP core that will not be used by the application. Eliminating logic that is guaranteed to not be used by an application can produce a design that has significantly lower area and power than the original microprocessor IP that targets an arbitrary application. As long as the approach to create a bespoke processor is automated, the resulting design retains the cost benefits of a microprocessor IP, since no additional hardware or software needs to be developed. Also, since no logic used by the application is eliminated, area and power benefits come at no performance cost. The resulting bespoke processor does not require programmer intervention or hardware support either, since the software application can still run, unmodified, on the bespoke processor.

6.1 Tailoring a Bespoke Processor

A bespoke processor, tailored to a target application, must be functionally-equivalent to the original processor when executing the application. As such, the bespoke implementation of a processor design should retain all the gates from the original processor design that might be needed to execute the application. Any gate that could be toggled by the application and propagate its toggle to a state element or output port performs a necessary function and must be retained to maintain functional equivalence. Conversely,

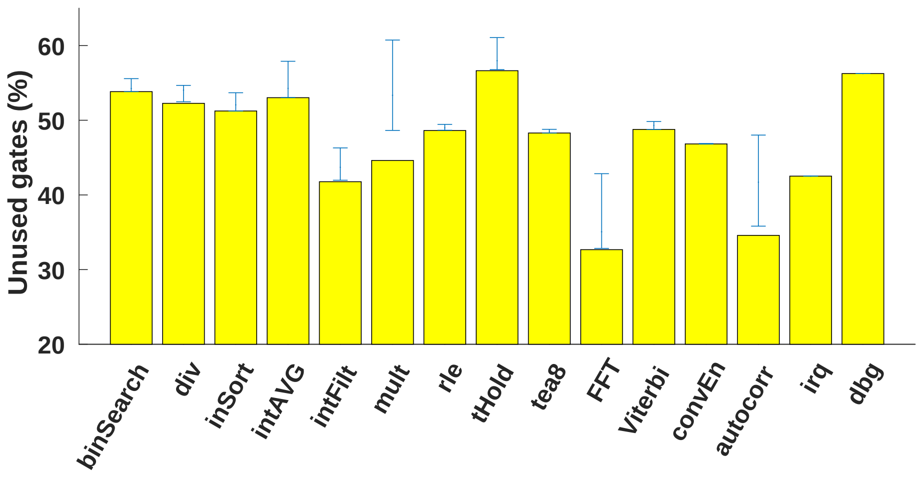


Figure 6.1: A significant fraction of gates in an openMSP430 processor are not toggled when an application executes on the processor. Each bar represents gates not toggled by any input for an application; the interval shows the range of unexercised gates for different inputs.

any gate that can never be toggled by the application can safely be removed, as long as each fanout location for the gate is fed with the gate's constant output value for the application. Removing constant (untoggled) gates for an application could result in significant area and power savings and, unlike conventional energy saving techniques, will introduce no performance degradation (indeed, no change at all in application behavior).

Figure 6.2 shows our process for tailoring a bespoke processor to a target application. The first step – *input-independent gate activity analysis* – is the proposed hardware-software co-analysis from Chapter 2. The second phase of our bespoke processor design technique – gate cutting and stitching – uses gate-level activity information gathered during gate activity analysis to prune away unnecessary gates and reconnect the cut connections between gates to maintain functional equivalence to the original design for the target application. Below we discuss the cutting and stitching process in detail.

6.1.1 Cutting and Stitching

Once gates that the target application cannot toggle have been identified, they are cut from the processor netlist for the bespoke design. After cutting out a gate, the netlist must be stitched back together to generate the final netlist and laid-out design for the

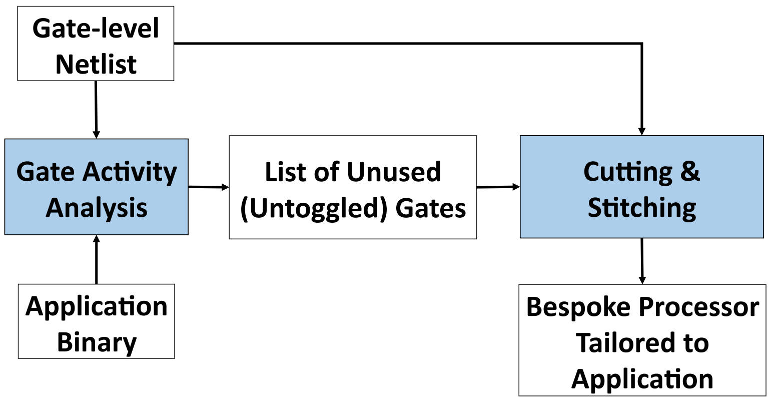


Figure 6.2: Our technique performs input-independent gate activity analysis to determine which gates of a processor cannot be toggled in any execution of the application. These gates are then cut from the design to form a custom, bespoke processor with reduced area and power.

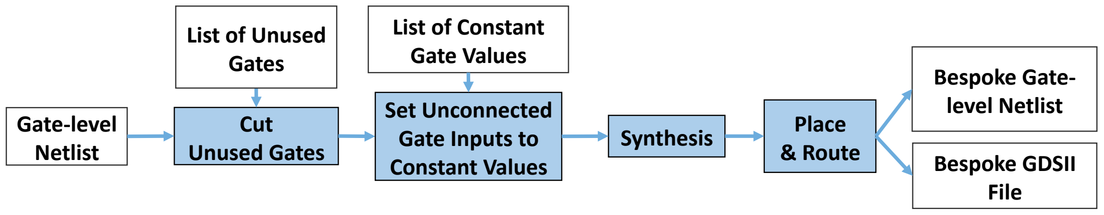
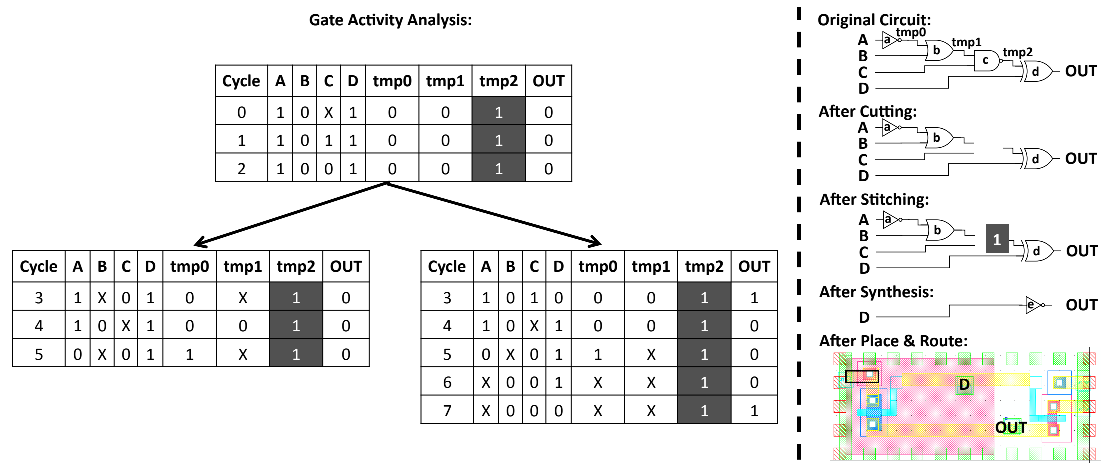

List of Unused Gates
List of Constant Gate Values
Gate-level Netlist
Cut Unused Gates
Set Unconnected Gate Inputs to Constant Values
Synthesis
Place & Route
Bespoke Gate-level Netlist
Bespoke GDSII File

Figure 6.3: Tool flow for cutting and stitching.

bespoke processor. Figure 6.3 shows our method for cutting and stitching a bespoke processor. First, each gate on the list of unusable (untoggled) gates is removed from the gate-level netlist. After removing a gate, all fanout locations that were connected to the output net of the removed gate are tied to a static voltage (‘1’ or ‘0’) corresponding to the constant output value of the gate observed during simulation. Since the logical structure of the netlist has changed, the netlist is re-synthesized after cutting all unusable gates to allow additional optimizations that reduce area and power. Since some gates have constant inputs after cutting and stitching, they can be replaced by simpler gates. Also, toggled gates left with floating outputs after cutting can be removed, since their outputs can never propagate to a state element or output port. Since cutting can reduce the depth of logic paths, some paths may have extra timing slack after cutting, allowing faster, higher power cells to be replaced with smaller, lower power versions of the cells.

Finally, the re-synthesized netlist is placed and routed to produce the bespoke processor layout, as well as a final gate-level netlist with necessary buffers, etc. introduced to meet timing constraints.

6.1.2 Illustrative Example



Cycle	A	B	C	D	tmp0	tmp1	tmp2	OUT
0	1	0	X	1	0	0	1	0
1	1	0	1	1	0	0	1	0
2	1	0	0	1	0	0	1	0

Cycle	A	B	C	D	tmp0	tmp1	tmp2	OUT
3	1	X	0	1	0	X	1	0
4	1	0	X	1	0	0	1	0
5	0	X	0	1	1	X	1	0

Cycle	A	B	C	D	tmp0	tmp1	tmp2	OUT
3	1	0	1	0	0	0	1	1
4	1	0	X	1	0	0	1	0
5	0	X	0	1	1	X	1	0
6	X	0	0	1	X	X	1	0
7	X	0	0	0	X	X	1	1

Figure 6.4: An example of gate activity analysis and cutting and stitching.

This section illustrates how bespoke processor design tailors a processor design to a particular application, as described in 6.1.1. Figure 6.4 illustrates the bespoke design process. The left part of Figure 6.4 shows input-independent gate activity analysis for a simple example circuit (top right). During symbolic simulation of the target application, logical 1s, 0s, and unknown symbols (Xs) are propagated throughout the netlist. In cycle 0, A and B have known values that are propagated through gates a and b, driving tmp0 and tmp1 to '0'. The controlling value at gate c drives tmp2 to '1', despite input C being an unknown value (X). Inputs A and B are not changed by the simulation of the binary until after cycle 2, when an X was propagated to the PC (not shown) that requires two different execution paths to be explored. In the left path, input B becomes X in cycle 3, causing tmp1 to become X as well. However, since input C is a '0', tmp2 is still a '1'. In the right execution path, inputs A and B both have Xs and logic values that may toggle tmp1 in cycles 5-7, but for each of these cycles, input C is a '0', keeping tmp2 constant at '1'. Since tmp2 is never toggled during any of the possible executions of the application, gate c is marked for cutting, and its constant output value (‘1’) is stored for stitching. Although gate d is never toggled in cycles 0-2 or down the left execution path, it does toggle in the right execution path and thus cannot be marked for cutting. Gates a and b also toggle and thus are not marked for cutting.

Once gate activity analysis has generated a list of cuttable gates and their constant values, cutting and stitching begins. Since gate c was marked for cutting, it is removed from the netlist, leaving the input to its fanout (d) unconnected. During stitching, d’s floating input is connected to c’s known constant output value for the application (‘1’). After stitching, the gate-level netlist is re-synthesized. Synthesis removes gates that are not driving any other gates (gates a and b), even though they toggled during symbolic simulation, since their work does not affect the state or output function of the processor for the application. Synthesis also performs optimizations, such as constant propagation, which replaces gate d with an inverter, since the constant controlling input of ‘1’ to the XOR gate makes it function as an inverter. Finally, place and route produces a fully laid-out bespoke design.

6.2 Correctness of operation

In this section, we show that the transformations we perform to create a bespoke processor implementation produce a design that is functionally equivalent to the original processor design for the target application. I.e., the bespoke design implements the same function and produces the same output as the original design for all possible executions of the application.

Theorem: A bespoke processor implementation $\mathcal{B}_{\mathcal{A}}$ of processor $\mathcal{P}$ tailored to an application $\mathcal{A}$ is functionally-equivalent to processor $\mathcal{P}$ with respect to application $\mathcal{A}$; $\mathcal{B}_{\mathcal{A}}$ produces the same output as $\mathcal{P}$ for any possible execution of $\mathcal{A}$.

Proof: The first step in creating $\mathcal{B}_{\mathcal{A}}$ – input-independent gate activity analysis – identifies the subset $\mathcal{E}$ of all gates in the processor that can possibly be exercised by $\mathcal{A}$, for all possible inputs. The analysis also identifies the constant output values for all gates $\mathcal{U}$ that can never be exercised by $\mathcal{A}$. It follows that $\mathcal{E} \cap \mathcal{U} = \emptyset$ and $\mathcal{E} \cup \mathcal{U} = \mathcal{G}$, where $\mathcal{G}$ is the set of all gates in $\mathcal{P}$. Cutting and stitching (Section 6.1.1) removes all gates in the set $\mathcal{U}$ and ties their output nets to their known constant values, such that the functionality of all gates in $\mathcal{U}$ is maintained in $\mathcal{B}_{\mathcal{A}}$. All gates in $\mathcal{E}$ remain in the bespoke design, so all gates in $\mathcal{E}$ have the same functionality and produce the same outputs in $\mathcal{B}_{\mathcal{A}}$ and $\mathcal{P}$. Since $\mathcal{E} \cup \mathcal{U} = \mathcal{G}$, it follows that $\mathcal{B}_{\mathcal{A}}$ is functionally equivalent to $\mathcal{P}$ for $\mathcal{A}$ and produces the same output as $\mathcal{P}$ for all possible inputs to $\mathcal{A}$. ■

6.3 Results

In this section, we evaluate bespoke processors. We first consider area and power benefits of tailoring a processor to an application, then evaluate design approaches that can be used to support bespoke processors throughout the product life-cycle, including procedures for verifying bespoke processors, techniques to design bespoke processors that support multiple known applications, and strategies to allow in-field updates in bespoke processors.

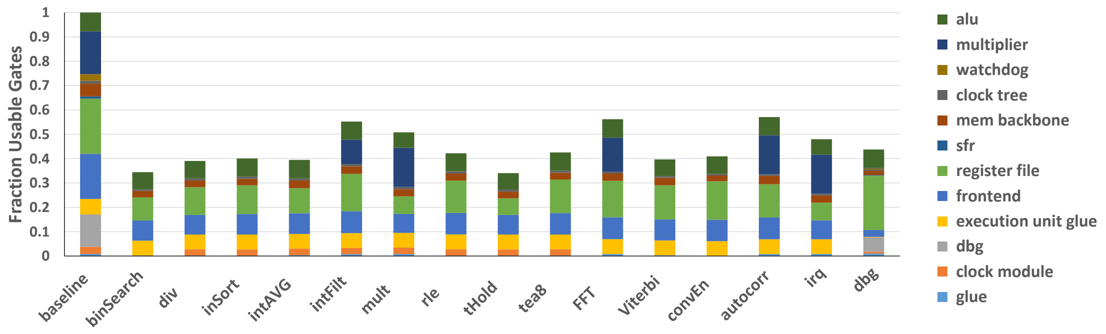


Figure 6.5: The height of a bar represents the fraction of gates that can be toggled by a benchmark. Components within each bar represent each module's contribution to the fraction of gates toggled by the benchmark.

Figure 6.5 shows the fraction of gates in the original processor design that could be toggled by each benchmark.[1] The components within each bar represent each module's contribution to the fraction of gates that can be toggled by the benchmark. The first bar in the figure shows each module's contribution to the total gates in the baseline design. We observe that each benchmark can toggle only a relatively small fraction of the gates in the baseline design. At most, 57% of the gates in the baseline design can

[1] Unlike Figure 6.1, which presents results from profiling, Figure 6.5 shows results from input-independent gate analysis.

be toggled, and 11 benchmarks toggle less than half the gates. Even though a large fraction of the gates of the baseline processor cannot be toggled by each benchmark, each benchmark can toggle a different set of gates. For example, `autocorr1`, which uses the largest fraction of the gates in the baseline processor, does not exercise the clock_module, while `tHold`, which toggles the smallest fraction of the baseline gates, does exercise gates in the clock_module.

Some modules, such as the multiplier, are used by some benchmarks and not others. However, module usage differs by application. For example, `intFilt` can never toggle about half of the multiplier gates due to constraints the binary places on filter coefficients, whereas `mult` toggles almost all the gates in the multiplier. Other modules, such as the frontend, are toggled by all applications, but each application can toggle a different subset of frontend gates. While these results show that a bespoke processor can have a significantly lower gate count than the general purpose processor it is derived from, they also confirm that hardware-software co-analysis is necessary to identify all the gates that can be eliminated in a bespoke design. Elimination of gates based on techniques such as profiling or static analysis will either fail to guarantee correctness or will miss opportunities to eliminate gates that an application can never use.

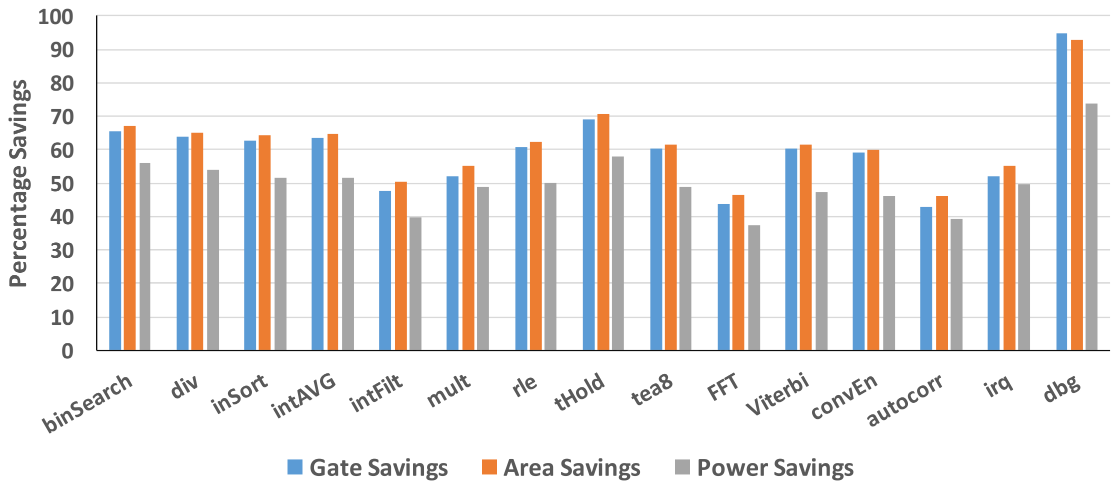


Figure 6.6: Reduction (%) in gate count, area, and power for a bespoke design, compared to the baseline processor.

Bespoke processors have fewer gates, lower area, and lower power than their general purpose counterparts. Figure 6.6 shows the reduction in gates, area, and power afforded

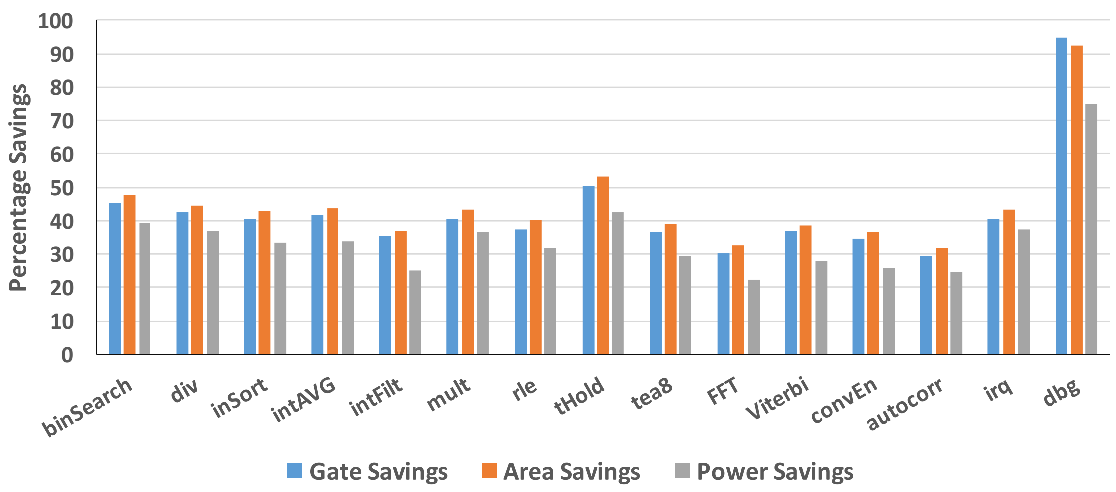


Figure 6.7: Reduction (%) in gate count, area, and power for bespoke designs, compared to application-specific coarse-grained module-level bespoke design.

by bespoke processors tailored to each benchmark. The benchmark `FFT`, which has the smallest gate count reduction (44%),[2] still reduces area by 47% and power by 37%, relative to the baseline design. Area savings are up to 92% (`dbg`), while power savings are up to 74% (`dbg`).

Figure 6.5 shows that some modules could be wholly removed for specific benchmarks (e.g., the multiplier can be removed for `binSearch`, since it cannot use any gates in the multiplier). For such modules, it is possible to use an Xtensa-like approach [89], enabled by our input-independent gate activity analysis, where modules in which no gates are usable by an application are removed from the design. Figure 6.7 shows the benefits of bespoke processors relative to coarse-grained bespoke designs in which wholly-unusable modules have been removed from the processor. Note that compared to an Xtensa-like approach, a coarse-grained bespoke design does not need any knowledge of the microarchitecture, as the unusable gates are identified automatically by hardware-software co-analysis. The results show that the fine-grained gate-level bespoke design can provide up to 75% power reduction (22% minimum, 35% on average) over coarse-grained module-level bespoke design.

Additional power savings may be possible when cutting, stitching, and re-synthesis

[2] Note that gate count reduction reported in Figure 6.6 is different than fraction of toggled gates in Figure 6.5, since bespoke design also removes some toggled gates that cannot propagate their toggles to state elements or output ports.

Table 6.1: Benefits of exploiting timing slack created by cutting, stitching, and re-synthesis.

Benchmark	**Timing Slack (%)**	**V$_{min}$ (V)**	**Addl. Power Savings from Slack (%)**	**Total Power Savings (%)**
binSearch	24.30	0.81	36.86	72.1
div	24.51	0.82	34.53	69.9
inSort	22.24	0.83	33.25	67.6
intAVG	23.37	0.83	33.35	67.7
intFilt	23.23	0.84	31.31	58.5
mult	20.45	0.91	20.33	59.3
rle	22.10	0.83	33.31	66.8
tHold	24.07	0.81	36.90	73.5
tea8	23.55	0.83	33.23	65.7
FFT	21.74	0.90	20.13	50.0
Viterbi	23.48	0.83	33.18	64.9
convEn	23.96	0.83	33.03	63.9
autocorr1	18.48	0.91	18.31	50.3
irq	17.91	0.92	16.24	57.7
dbg	45.70	0.60	67.73	91.5

removes gates from critical paths, exposing additional timing slack that can be exploited for energy savings. Table 6.1 shows timing slack exposed during bespoke processor tailoring for each benchmark. Exposed timing slack can be used to reduce the operating voltage of the processor without reducing the frequency.[3] Table 6.1 also shows the minimum safe operating voltage for each bespoke design (assuming worst-case PVT variations), the additional power savings afforded by exploiting timing slack in bespoke designs, and the total power savings achieved with respect to the baseline design from eliminating unusable logic and exploiting exposed timing slack for voltage reduction.

6.3.1 Supporting Multiple Programs and In-field Updates

Bespoke processors are able to support multiple programs by including the union of gates needed to support all of the programs. Figure 6.8 shows gate count, power, and area for bespoke processors tailored to N programs, normalized to the baseline processor. For each value of N, the bars show the ranges of these metrics across bespoke

[3] Exposed timing slack could also be used to increase operating frequency (performance) of a bespoke design. On average, frequency can be increased by 13% in the bespoke designs.

processors tailored to all combinations of N programs. For many combinations of programs, even for up to ten programs, only 60% or less of the gates are needed. In fact, despite supporting ten programs, the area and power of a bespoke processor can be reduced by up to 41% and 20%, respectively. However, supporting multiple programs can limit the extent of gate cutting and the resulting area and power benefits when the applications exercise significantly different portions of the processor. For example, the two-program bespoke processor with the largest gate count is one tailored to `dbg` and `irq`. Each application uses components of the processor that are not exercised by the other program; `dbg` exercises the debug module, while `irq` exercises the interrupt handling logic. The resulting gate reduction of 18% still produces area and power benefits of 26% and 19%, respectively. Although supporting multiple programs reduces gate count, area, and power reduction benefits, the area and power will never increase with respect to the baseline design. In the worst case, the baseline processor can run any combination of program binaries.

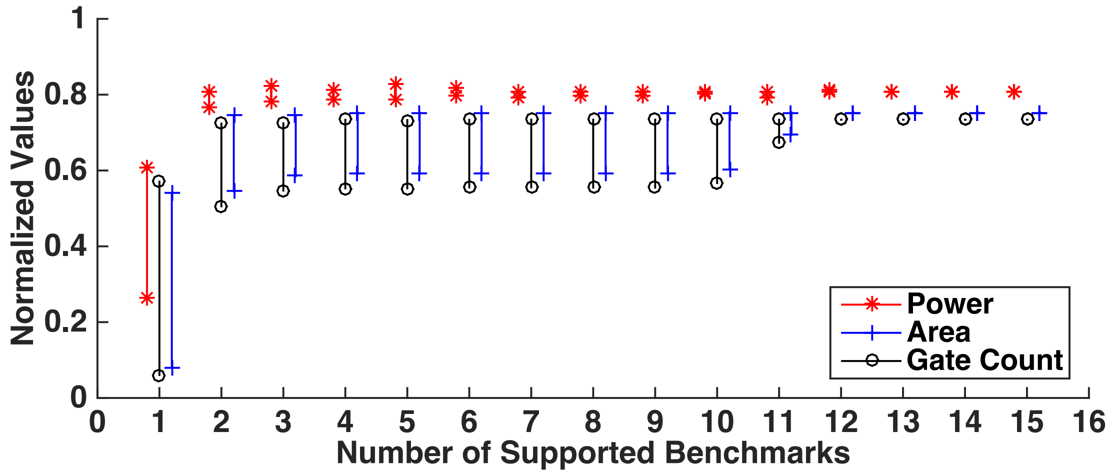


Figure 6.8: Normalized gate count, area, and power ranges for all possible bespoke processor supporting multiple applications.

We consider two approaches to designing bespoke processors that can be updated in the field. First, we evaluate a method that allows a bespoke processor to handle common, minor programming bugs. Second, we evaluate a method that allows a bespoke processor to handle infrequent, arbitrary software updates.

Table 6.2: Milu produces three types of mutants. Type I: Logical conditional operator mutants. Type II: Computation operator mutants. Type III: loop conditional operator mutants.

Benchmark	**Type I**	**Type II**	**Type III**	**Total**
binSearch	0	0	15	15
inSort	8	0	15	23
rle	0	20	25	45
tea8	48	24	10	82
Viterbi	24	24	35	83
autocorr	12	0	10	22

In-field updates may often be deployed to fix minor correctness bugs (e.g., off-by-one errors, etc.) [90]. To emulate in-field updates to fix bugs, we use the Milu mutation testing tool [91] to generate "updates" corresponding to bug fixes. Table 6.2 lists the breakdown of mutants by type generated by Milu for the six benchmarks with the most mutants. If a benchmark has zero mutants for a particular type, no mutation sites of that type were found in that benchmark by Milu. Type I mutants are conditional operator mutants (e.g., $A||B \rightarrow A\&\&B$). Type II mutants are computation operator mutants (e.g., $A+B \rightarrow A\times B$). Type III mutants are loop conditional operator mutants (e.g., $i < 32 \rightarrow i \neq 32$).

Table 6.3: Percentage of mutants (in-field updates) of different types that are supported by the bespoke design for the base software implementation. "-" denotes that a given benchmark did not have any mutants of that type.

Benchmark	**Type I %**	**Type II %**	**Type III %**	**Total %**
binSearch	-	-	73	73
inSort	25	-	27	26
rle	-	100	84	91
tea8	58	75	100	68
Viterbi	92	83	80	84
autocorr	50	-	40	45

Table 6.3 lists the percentage of mutants (i.e., in-field updates to fix bugs) that are supported by the original bespoke design (generated for a "buggy" application). Many minor bug fixes can be covered by a bespoke processor designed for the original

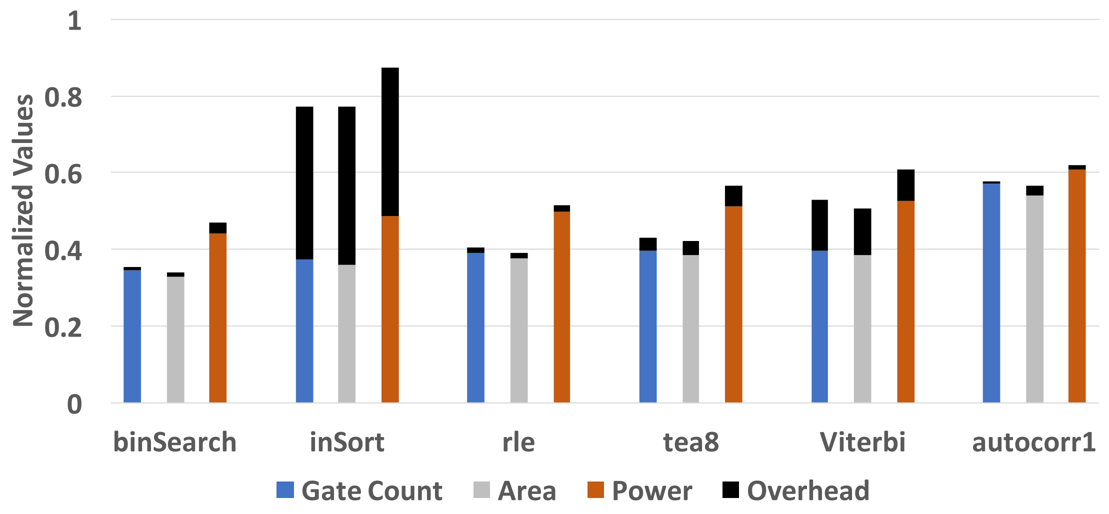


Figure 6.9: Normalized gate count, area, and power vs the baseline design for designs supporting all mutants (in-field updates).

application without any modification. I.e., the mutants representing many in-field updates only use a subset of the gates in the original bespoke processor. This means that these mutants will execute correctly on the original bespoke processor tailored to the original "buggy" application. We see that between 25% and 100% of various mutants are covered, and 70% of all mutants are covered. This shows that a bespoke processor will maintain some of the original general purpose processor's ability to support in-field updates. If a higher coverage of possible bugs is desired, the automatically-generated mutants can be considered as independent programs while tailoring the bespoke processor for the application. Figure 6.9 shows the increase in gate count, area, and power required to tailor a bespoke processor to the six benchmarks with the most mutants by including all possible mutants (i.e., bug fixes) generated by Milu during bespoke design. Providing support for simple in-field updates incurs a gate count overhead of between 1% and 40%. Despite this increase in gate count, total area benefits for the bespoke processors are between 23% and 66%, while total power benefits are between 13% and 53%. Therefore, simple in-field updates can be supported while still achieving substantial area and power benefits.

A bespoke processor tailored to a specific application can be designed to support arbitrary software updates by designing it to support a Turing-complete instruction (e.g., `subneg`) or set of instructions, in addition to other programs it supports. For our single-application bespoke processors, the average area and power overheads to support `subneg` are 8% and 10%, respectively. Average area and power benefits for `subneg`-enhanced bespoke processors are 56% and 43%, respectively.

Note that an instruction in a bespoke processor's target application is not guaranteed to be supported in a different application (e.g., an update), since the processor eliminates gates that are not needed to support the possible instruction sequences in the target application's execution tree; a different sequence of the same instructions may need those gates for execution. For example, if all operands to `add` instructions in a bespoke processor's target binary have had their least significant eight bits masked to 0 by a preceding `and` instruction, gates corresponding to the least significant bits of the ALU's adder may be removed in the bespoke processor. Therefore, the same bespoke processor may not support a different program where the `add` instruction is not preceded by the masking `and`. While full support for instructions is not guaranteed in general by bespoke processors, we are able to guarantee support for Turing-complete instructions / instruction sequences (e.g., `subneg`), since a software routine written using a Turing-complete instruction / instruction sequence consists entirely of multiple instances of the same instruction / instruction sequence.

6.3.2 System Code

The evaluations above were performed for a bare-metal system (application running on the processor without an operating system (OS)). While this setting is representative of ultra-low-power processors and a large segment of embedded systems [92, 93],[4] use of an OS is common in several embedded application domains, as well as in more complex systems. Thus, we also evaluated bespoke design for our applications running on the processor with an OS (FreeRTOS [96]). Application analysis of system code for FreeRTOS reveals that 57% of gates are not exercisable by the OS, including the entire hardware multiplier. When our benchmarks are evaluated individually with FreeRTOS, 37% of gates are unused in the worst case, 49% on average. When running FreeRTOS together with all 15 benchmarks, 27% of gates are unused.

[4] Many embedded processors provide bare-metal development toolchains [94, 95].

Chapter 7

Software-based Gate-level Information Flow Security

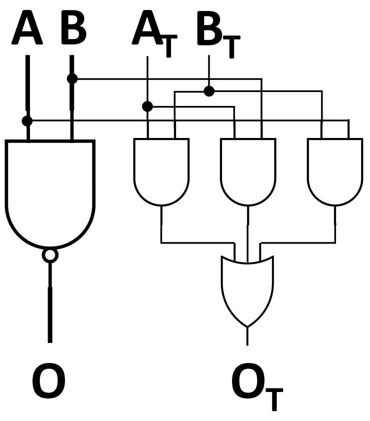
In this chapter we show how the hardware-software co-analysis proposed in Chapter 2 can be used to track information flow of an application at the gate-level of an ultra-low-power embedded microprocessor. As the internet of things progresses toward the internet of everything, higher connectedness implies more security attack vectors and a larger attack surface. In the last couple of years, reported IoT attacks include compromising baby monitors to enable unauthorized live feeds, interconnected cars to control a car in motion, smart watches and fitness trackers to steal private information and health data, power grids and steel mills to render them offline, and medical devices with detrimental, perhaps fatal, consequences on patients' health. Consequently, security and privacy have become first order design concerns for IoT systems. However, IoT systems are often ill-protected, in spite of their critical security implications, due to their limited energy and area budget to spend on security.

Prior work on gate-level information flow tracking shows that information security guarantees can be provided through techniques that track information flows at the gate level, but unfortunately, such solutions rely on non-commodity, secure-by-design processors; the ultra-low power and area constraints of ULP systems may make such approaches infeasible. However, we have observed that many of the architectural changes required in existing secure-by-design processors arise because prior works assume that all software running on a system besides the kernel is completely unknown, in order to provide a security guarantee for all applications. Since the application running on a ULP system is often simple, we have introduced application-specific information flow tracking that takes all of a system's software into consideration during security analysis, enabling gate-level information flow security guarantees for commodity systems, without the programmability, performance, and monetary costs of a specialized secure-by-design processor. Our software-based solution to gate-level information flow tracking identifies all possible insecure information flows for the software running on a system, as well as the instructions that can cause violations, and allows security vulnerabilities to be eliminated through software modifications. By targeting only the vulnerabilities that an application is susceptible to, we minimize, and in some cases completely eliminate, the overhead of guaranteeing security for ULP systems.

7.1 Information Flow Security

Information flow security aims to (1) determine if any information flows exist from one state element (e.g., a variable in a program) to another state element and to (2) prevent or warn users of such flows when a flow violates an *information flow policy.* Past work [97, 98, 99, 100, 101, 102] has performed information-flow tracking at the software level and demonstrated its effectiveness at detecting a set of security vulnerabilities without modification of the hardware (i.e., applicable on commodity hardware). Other work [103, 104, 105, 106] proposes hardware modifications for improved efficiency and accuracy of ISA-level information flow tracking. Unfortunately, these approaches not only require hardware modifications, but they may still miss information flows that crop up as a result of the low-level implementation details of a processor [107]. Our approach aims to achieve the advantages of both software-based and hardware-based information flow tracking – applicability to unmodified commodity hardware, accuracy in tracking information flows, and minimal runtime overhead – without the corresponding limitations.

In order to track all forms of digital information flow, Tiwari *et al.* [107] proposed gate-level information flow tracking (GLIFT). As shown in Figure 7.1, GLIFT augments



A	A_T	B	B_T	O	O_T
0	0	0	0	1	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	1	1
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	1	0	1

Figure 7.1: Example truth table for gate-level information flow tracking of a NAND gate. A '1' in the taint value columns (shaded gray) represents a tainted value (e.g., untrusted or secret values).

each gate in a design with taint-tracking hardware. The taint of a gate's output is determined by the values and taints of its inputs.By propagating taint values through each gate, tainted data (e.g., untrusted or secret) can be tracked from input ports (or other marked data, including instructions in program memory) through the processor at the gate level to guarantee that no tainted data reaches an output port that should remain untainted (e.g., a trusted or non-secret output). When fabricated with the base design, GLIFT can dynamically track taints at a high degree of accuracy, albeit at up to a 3× overhead in hardware. More recently, GLIFT has been used to statically track information flows [108]. In this work, an analysis called *-logic is used to statically track taints for a microkernel with no non-determinism running on hardware designed to be easily verifiable. The focus was on performing gate-level information flow tracking for a specific, application-agnostic secure-by-design system. We focus, instead, on performing application-specific gate-level information flow tracking for arbitrary IoT applications on commodity hardware, including applications with control dependencies on unknown, tainted inputs. When analyzed with *-logic, such applications could unnecessarily taint all software-exercisable gates

Based on the insights and verification of GLIFT, several secure-by-design processors have been built. They range from a predication-based, non-Turing-complete processor [107] to processors that can handle arbitrary computations through hardware compartmentalization [109, 108]. While these processors can guarantee that any software that runs on them cannot violate a non-interference information security policy (i.e., no untrusted inputs can affect trusted outputs and no secret inputs can affect non-secret outputs), they can be limited in their programmability (e.g., [107] requires all loops to be statically bounded while [109] does not naturally support unbounded or variable-length operations) and require hardware modifications (e.g., partitioned memory structures and memory bounds checking hardware). The cost of any re-design of a commodity microcontroller may be prohibitive. In this paper, we design full systems that ensure the same non-interference policy as [108] (i.e., no untrusted input can affect a trusted output and no secret input can affect a non-secret output), but on a per-application basis.

Recently, a body of work has emerged on developing hardware description languages and tools to design and verify information flow secure hardware [110, 111, 112, 113]. While such works can prove that a hardware design meets an information flow security policy, even one that is commercial, such as ARM's Trustzone [113], these approaches cannot verify commodity hardware that does not already implement information flow security. Our approach targets commodity hardware, in addition to emerging hardware, and allows application developers to demonstrate the security of their applications at a fine-grained level.

7.2 Application-Specific Gate-Level Information Flow Tracking

Performing gate-level information flow tracking that is application-specific comes with several challenges. While it does allow secure-by-design systems to be built on commodity hardware, it requires a means of identifying all possible insecure information flows that may occur in a system, for all possible executions of the system's software, for any possible inputs that may be applied to the system. In this section, we describe an automated technique that takes as input the hardware description (gate-level netlist) of a processor, the software that runs on the system, and labels identifying trusted / untrusted (or secure / insecure) inputs and outputs in the system and efficiently explores all possible application-specific execution states for the system to identify all possible insecure information flows in the system. The output from our automated framework can be used to verify the information flow security of a system as well as to guide and automate software modification to eliminate information flow security vulnerabilities in the system.

Figure 7.2 shows the process for verifying a security policy using application-specific gate-level information flow tracking. The first step performs offline input-independent gate-level taint tracking of an entire systems binary running on a gate-level description of a processor. The initial components that are tainted are specified by the information flow security policy (e.g., ports labeled as untrusted or memory locations labeled as secret). The result of taint tracking is a per-cycle representation of tainted state (both gates and memory bits). The second step performs information flow policy checking where the information flow checks specified by the information flow security policy are verified on the per-cycle tainted state. The result is a list of possible violations of the information flow security policy.

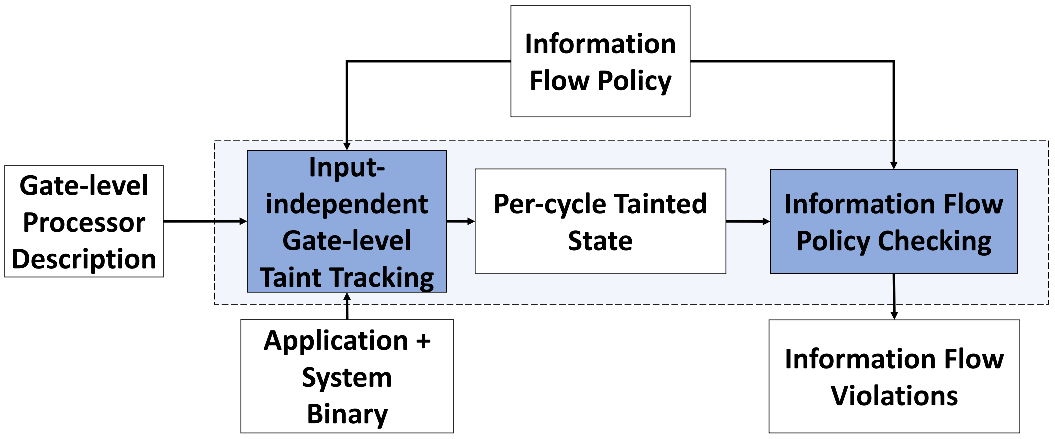

Information Flow Policy
Gate-level Processor Description
Input-independent Gate-level Taint Tracking
Per-cycle Tainted State
Information Flow Policy Checking
Application + System Binary
Information Flow Violations

Figure 7.2: Application-specific gate-level information flow tracking evaluates specific information flow security policies across all possible executions of the entire system binary, producing a list of all possible violations.

7.2.1 Input-independent Gate-level Taint Tracking

Algorithm 3 describes our input-independent gate-level taint tracking. It is an augmented version of Algorithm 1 described in Chapter 2 where net values also maintain taint information. Initially, the values of all memory cells and gates are set as unknown values (i.e., Xs) and are marked as untainted. The system binary, consisting of both tainted and untainted partitions,[1] is loaded into program memory. Our tool performs input-independent taint tracking based on symbolic simulation, where each bit of an input is set to an unknown value symbol, X. Additionally, inputs or state elements may be tainted according to the specified information flow security policy (e.g., the non-interference policy described in Section 7.1). Throughout simulation, logical values are propagated throughout the circuit as standard ternary logic. Taint values, which are dependent on both the taint values of inputs and their logical values, are propagated as described in [107] and exemplified in Figure 7.1.

A key difference between our input-independent gate-level taint tracking and prior analyses such as *-logic occurs when an unknown symbol propagates to the PC. For example, directly applying a *-logic analysis on commodity hardware to an application where the PC becomes unknown and tainted results in most of the gates in the hardware also becoming unknown and tainted, since most gates are impacted by the PC. However, in our analysis, if an X propagates to the PC, indicating input-dependent control flow, our simulator branches the execution tree and simulates execution for all possible branch paths, following a depth-first ordering of the control flow graph. This combined with maintaining conservative state at each branch (described in Chapter 2) ensures scalability of our technique.

The result of the simulation is a pruned execution tree that stores both the logical and taint values at each point. Due to the conservative approximation technique (see Chapter 2) we can perform input-independent gate-level taint tracking in a tractable amount of time, even for applications with an exponentially-large or infinite number of execution paths.[2]

[1] Note that tainted and untainted code partitions do not indicate that the corresponding instructions are marked as tainted or untainted in the program memory, although our tool allows them to be.

[2] Some complex applications and processors might still require heuristics for exploration of a large number of execution paths [22, 23]; however, our approach is adequate for ultra-low-power systems,

Algorithm 3 Input-independent Gate-level Taint Tracking

```
1.  Procedure Taint Tracking(system_binary, design_netlist, security_policy)
2.  Initialize all memory cells and all gates in design_netlist to untainted X
3.  Mark tainted ports and gates according to security_policy
4.  Load system_binary into program memory
5.  Propagate reset signal
6.  $s \leftarrow$ State at start of system_binary
7.  Table of previously observed symbolic states, $T$.insert($s$)
8.  Symbolic execution tree, $S$.set_root($s$)
9.  Stack of un-processed execution points, $U$.push($s$)
10. mark_all_gates_untoggled(design_netlist)
11. while $U$ != $\emptyset$ do
12.     $e \leftarrow U$.pop()
13.     while $e$.PC_next != X and !$e$.END do
14.         $e$.set_inputs_X() // set all peripheral port inputs to Xs
15.         $e$.set_taints(security_policy) // taint appropriate state according to security_policy
16.         $e' \leftarrow$ propagate_gate_values($e$) // simulate this cycle
17.         $t \leftarrow$ propagate_taint_values($e'$,$e$) // determine taint values for e'
18.         $S$.add_simulation_point($e'$,$t$) // store logical and taint state
19.         if $e'$.modifies_PC then
20.             $c \leftarrow T$.get_conservative_state($e$)
21.             if $e' \not\subset c$ then
22.                 $T$.make_conservative_superstate($c$,$e'$)
23.             else
24.                 break
25.             end if
26.         end if
27.         $e \leftarrow e'$ // advance cycle state
28.     end while
29.     if $e$.PC_next == X then
30.         $c \leftarrow T$.get_conservative_state($e$)
31.         if $e \not\subset c$ then
32.             $e' \leftarrow T$.make_conservative_superstate($c$,$e$)
33.             for all $a \in$ possible_PC_next_vals($e'$) do
34.                 $e'' \leftarrow e$.update_PC_next($a$)
35.                 $U$.push($e''$)
36.             end for
37.         end if
38.     end if
39. end while
```

7.2.2 Information Flow Checking

The result of input-independent gate-level taint tracking is a conservative symbolic execution that represents all possible executions of the entire system's binary. This symbolic execution tree is annotated with logical gate values and associated taint values. Using these taint values, information flow checking can be performed where the specific security policy is checked. An example information flow security policy is defined by [108]: input and output ports are labeled as trusted or untrusted and, independently, as secret

representative of an increasing number of future uses which tend to have simple processors and applications [24, 11]. For example, complete analysis of our most complex system takes 3 hours.

or non-secret (i.e., untrusted and secret are two taints that are analyzed separately). An attacker is assumed to have complete control over all untrusted inputs to the device and controls the initial implementation of untrusted code, which is known at analysis time. No untrusted information can flow out of a trusted port, and no secret information can flow out of a non-secret port.

7.3 Guaranteeing Information Flow Security

In this section, we describe software-based techniques that eliminate information flow security vulnerabilities in applications. Section 7.3.1 establishes conditions that are sufficient to guarantee information flow security, and Section 7.3.2 describes software transformations that are designed to guarantee that an application that is vulnerable to insecure information flows will satisfy the sufficient conditions. In Section 7.3.3, we prove that the transformations satisfy the sufficient conditions and ensure information flow security.

7.3.1 Sufficient Conditions that Guarantee Information Flow Security

In this section, we lay out a set of conditions that are sufficient for guaranteeing the non-interference information flow security policy described in Section 7.1. Later, we will show how our application-specific approach to information flow security satisfies these conditions.

(1) All processor state elements are untainted before untainted code (i.e., trusted or non-secret code) is executed.
(2) Tainted code does not taint an untainted memory partition used by untainted code.
(3) Untainted code does not load data from a tainted memory partition.
(4) Untainted code does not read from tainted input ports.
(5) Tainted code does not write to untainted output ports.

While the conditions above are not necessary for guaranteeing information flow security, they are sufficient; i.e., a system that maintains the conditions will not leak information. For an information leak of tainted data to occur, tainted data must be accessible to an untainted task in some state or memory element or through a port; a leak occurs when an untainted task propagates accessible tainted data to an untainted output that it has access to, or when a tainted task sends tainted data directly to an untainted output. The conditions above are sufficient to guarantee information flow security because they preclude all possible direct (through a port) or indirect (through state or memory) channels through which tainted information could leak. The first four conditions preclude all possible indirect information flows of tainted data, stating that if an untainted task executes in a taint-free processor, its memory partition remains taint-free, and it does not load tainted data from tainted memory or ports, its computations and outputs will remain untainted. The last condition precludes direct information flows of tainted data, stating that a tainted task is not allowed to write to untainted output ports.

Since the set of conditions above are sufficient, a system that meets the conditions guarantees non-interference. Secure-by-design processors use hardware-based information flow control mechanisms to guarantee that the above conditions are met for all possible applications that execute on the processor [107, 109, 108]. However, none of the conditions above are actually *necessary* to guarantee non-interference. For example, it is acceptable for state elements to be tainted when an untainted task executes (a violation of condition 1), as long as the computations performed by the task do not depend on any tainted state elements. Similarly, exceptions can be made for all the sufficient conditions (they are not necessary). Thus, as long as the original non-interference property (see Section 7.1) holds, any or all of the sufficient conditions described above may be relaxed. This insight has several interesting implications. (1) Since our symbolic analysis technique for input-independent gate-level taint tracking can check whether the non-interference property holds for all possible executions of a known application without forcing the application to meet the conditions above, it is possible to provide a security guarantee for any application that has no possible violations, even on a commodity processor that is not secure by design. (2) Since symbolic input-independent gate-level taint tracking can identify all possible instances where an application causes the non-interference property to be violated for a system, it can be used to identify locations where an application must be modified to prevent insecure information flows, as well as to verify whether a modified application is secure. (3) Some applications have no possibility of violating one or more of the conditions above. Therefore, some security mechanisms applied by secure-by-design processors represent unnecessary overhead for those applications. On the other hand, if insecure information flows can be eliminated through software modifications, the modifications can specifically target only the insecure information flows to which an application is vulnerable, potentially reducing the overhead of providing security for the system and enhancing programmability (by imposing fewer restrictions on software).

7.3.2 Software Techniques to Eliminate Insecure Information Flows

When the sufficient conditions for information flow security described in the previous section are not satisfied, it is possible for tainted information to leak. For example, allowing an untainted task to read and operate on tainted data may result in tainting of a processor's control flow state, and subsequently the execution of an untainted task. Specifically, if a processor's program counter (PC) becomes tainted, then all subsequent instructions will be tainted. Therefore, the control flow of an untainted computational task can also become tainted if it executes after a tainted task that taints the processor's control flow state. In fact, once the PC is tainted by a tainted task, it is possible that control may never become untainted, even if control is returned to untainted code. Preventing information flows from tainted to untainted code must include prevention of all direct information flow (e.g., the tainted code cannot call a yield function to return to untainted execution) and all indirect information flow (i.e., there must exist a mechanism that deterministically bounds the execution time of the tainted code). To avoid information leaks through control flow, there must exist an untaintable, deterministic mechanism that recovers the PC to an untainted state that fetches code from an untainted code partition.

Another common way for information to leak in a commodity processor is through the memory. If code that is allowed to handle tainted information writes to data memory using a fully tainted address, then the entire data memory, including partitions belonging to untainted code, will become tainted. For example, if tainted code reads a value from a tainted input port and then uses the value as an index to write into an array, the tainted address causes the entire data memory to become tainted, not just the memory location pointed to by the address. To avoid such leaks, a mechanism is needed to guarantee that no possible execution of tainted code can write to an untainted data memory partition.

For cases where an application violates the sufficient conditions and is vulnerable to insecure information flows, we propose two software transformations, analogous to the hardware mechanisms presented in [109], that target and prevent insecure information flows.

Untainted Timer Reset: An untainted timer can be used to reset the PC to an untainted location after a deterministic execution time of running tainted code, thus guaranteeing that tainted code cannot affect the execution of untainted code. However, on a commodity processor (e.g., openMSP430), generating such a timer is challenging for two reasons. First, common mechanisms for setting the PC, such as interrupts, still depend on the current, possibly tainted state of the pipeline to determine when the PC is reset. Second, the timer must not become tainted. As an example, on the openMSP430, a timer could be directly tainted by tainted code writing to its memory-mapped control register. To overcome these challenges, we propose using the watchdog timer that is common to many microcontrollers to reset the entire processor after a deterministic-length period of tainted execution. We use our symbolic simulation-based analysis to guarantee that the watchdog remains untainted.

Figure 7.3 shows our proposed watchdog timer reset. During the execution of a context switch in an untainted system code partition, the watchdog timer is set to a predetermined value for the computational task that is being switched in. The untainted system code then transfers execution to the tainted computational task. This tainted task can make full use of the processor, except writing to the watchdog or an untainted memory space partition or port, possibly propagating taints throughout the pipeline. When the untainted watchdog expires, it resets the entire pipeline with a power-on reset (POR).[3] Since this reset is untainted, the state within the pipeline will be reset to untainted values, including the PC. While using the watchdog timer flushes tainted data from the processor, the subsequent reset state is only untainted if the watchdog timer itself remains untainted. Since applications are known during analysis, the symbolic simulation used during input-independent gate-level taint tracking allows us to identify whether or not any tainted code can write to the control register of the watchdog timer during any possible execution of the tainted code. If there is no possibility of tainted

[3] We assume that the POR does not reset memory. This is a reasonable assumption, since many microcontrollers have non-volatile memory, including TI's MSP430FRXX series.

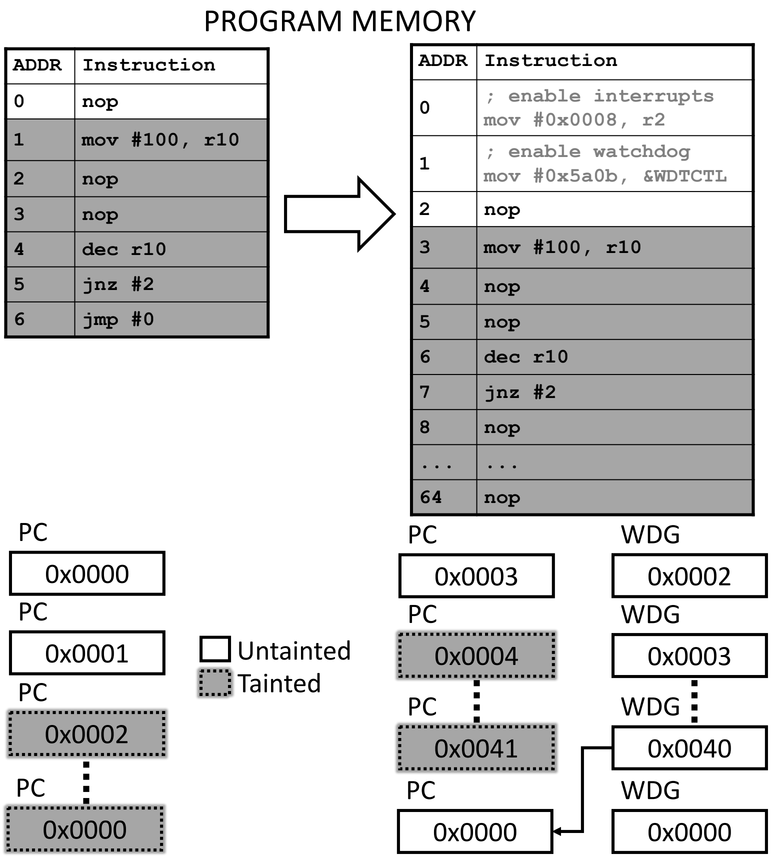


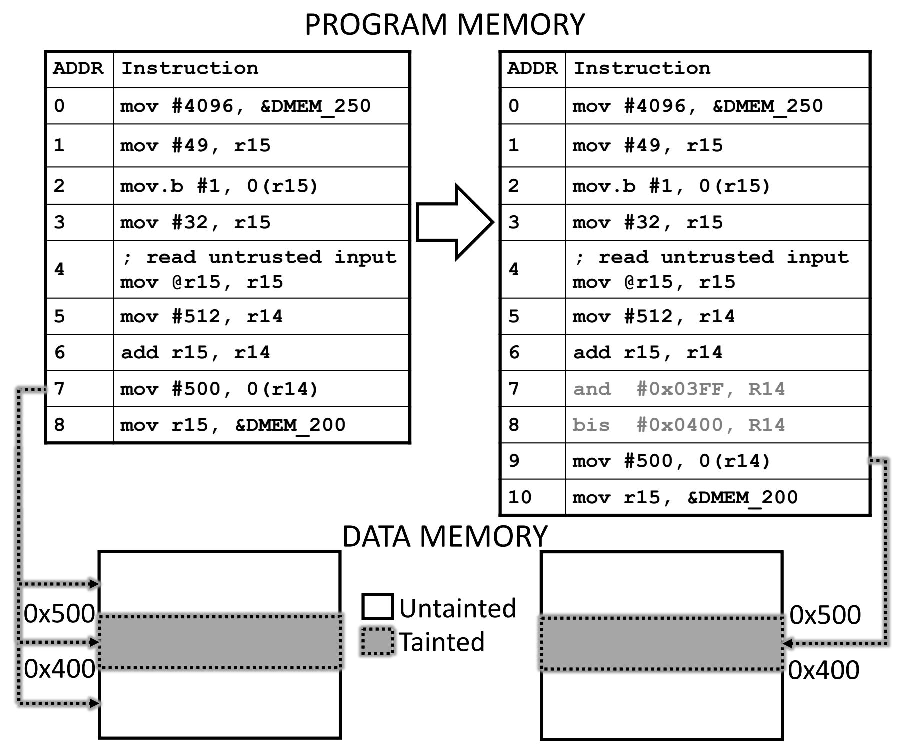
Figure 7.3: Untainted timer reset example: by starting a watchdog timer in the untainted portion of the code we ensure that the PC becomes untainted after the watchdog (WDG) issues a reset, restoring the system to an untainted (or trusted) state.

code writing to the control register of the watchdog timer, the write enable input for the control register is verified to be untainted. The only information this can leak is the fact that the tainted code does not access the watchdog timer – a known requirement for guaranteeing information flow security using our approach.

Note that this mechanism works naturally in multi-programming and task switching environments that are common in realtime embedded systems. Before context switching to a tainted computational task, the untainted system code simply sets the watchdog timer to the appropriate interval for the task – either the maximum length of the task or the length of an OS time slice, depending on the usage scenario. Expiration of the timer resets the processor to an untainted state, as usual, which also resets the PC. The code at PC=0 either contains or vectors to the system routine for switching in the next context.

If a tainted computational task wants to use the watchdog timer, it may not be possible to certify the system as *secure* unless a) it is impossible for the tainted task to cause a control flow violation or b) an alternative, functionally-equivalent (or otherwise acceptable) option can be used in place of the watchdog timer. Microprocessors typically provide several hardware timers, and it may be possible to emulate the functionality desired by the tainted task using a different timer. If it is not possible to use another available timer, software optimizations such as prediction may be used to eliminate the possibility of control flow violations.

Software Masked Addressing:



ADDR	Instruction
0	`mov #4096, &DMEM_250`
1	`mov #49, r15`
2	`mov.b #1, 0(r15)`
3	`mov #32, r15`

4	`; read untrusted input`
	`mov @r15, r15`

5	`mov #512, r14`
6	`add r15, r14`
7	`mov #500, 0(r14)`
8	`mov r15, &DMEM_200`

ADDR	Instruction
0	`mov #4096, &DMEM_250`
1	`mov #49, r15`
2	`mov.b #1, 0(r15)`
3	`mov #32, r15`

4	`; read untrusted input`
	`mov @r15, r15`

5	`mov #512, r14`
6	`add r15, r14`
7	`and  #0x03FF, R14`
8	`bis  #0x0400, R14`
9	`mov #500, 0(r14)`
10	`mov r15, &DMEM_200`

Figure 7.4: Memory mask example: By adding two instructions to addresses 7 and 8 in the right-hand code that mask the address to only use the tainted task's memory partition, no untainted memory locations get tainted.

Figure 7.4 shows our proposed memory bounds masking. The left side shows the original assembly code where a tainted address is used to store data, tainting the entire data memory. On the right side, the assembly code is modified to mask the memory address to guarantee that it falls within the region of data memory to which tainted code is allowed to write. Input-independent taint tracking can then verify that no taint is propagated to memory regions that are untainted. While simple masking solves the memory address taint problem for the case where the PC remains untainted, masking alone cannot guarantee information flow security when the PC becomes tainted. In this case, the tainted PC taints the masking instructions themselves. However, during application-specific gate-level information flow tracking, the program, including the added masking instructions, is known. In this case, our information flow tracking analysis can verify that no possible execution of the tainted code can generate an address outside of the regions of data memory that are allowed to be tainted. If there is no possibility of being able to write outside of allowed memory regions, there is no possibility of information flow, either explicit or implicit, between the allowed and disallowed memory regions. The only information flow that can leak is the information that the tainted application does not write outside of its allowed memory region – a known condition for guaranteeing information flow security.

7.3.3 Proving Information Flow Security

Theorem: For a system $\mathcal{S}$ consisting of a processor $\mathcal{P}$ and application $\mathcal{A}$, if application-specific gate-level information flow tracking $\mathcal{T}_{\mathcal{S}}$ of $\mathcal{S}$ reports that $\mathcal{S}$ is secure (i.e., satisfies the non-interference property), tainted data in $\mathcal{P}$ will never influence the execution of a trusted computational task $\mathcal{A}_{\mathcal{I}}$ in $\mathcal{S}$, and $\mathcal{P}$ will never propagate tainted data through an untainted output.

Proof: For tainted data to influence the execution of $\mathcal{A}_{\mathcal{I}}$, a taint must propagate from a tainted input of $\mathcal{S}$ to an untainted output written by $\mathcal{A}_{\mathcal{I}}$ either through a state element of $\mathcal{P}$, through the memory, or directly from a port.

Case 1 – taint propagation through a state element: For taintedness to influence $\mathcal{A}_{\mathcal{I}}$ through a state element $\mathcal{E}$, $\mathcal{E}$ must be tainted by a tainted computational task $\mathcal{A}_{\mathcal{J}}$ and remain tainted while $\mathcal{A}_{\mathcal{I}}$ is executing on $\mathcal{P}$. However, in any case where $\mathcal{T}_{\mathcal{S}}$ identifies a possible tainted information flow from $\mathcal{A}_{\mathcal{J}}$ to $\mathcal{A}_{\mathcal{I}}$, $\mathcal{A}$ is modified to invoke the watchdog timer mechanism to reset all state elements in the processor after the execution of $\mathcal{A}_{\mathcal{J}}$ and before the execution of $\mathcal{A}_{\mathcal{I}}$. Therefore, taint propagation through a state element is impossible, as long as $\mathcal{A}_{\mathcal{J}}$ does not interfere with the untainted operation of the watchdog timer. Since $\mathcal{T}_{\mathcal{S}}$ checks all possible execution states of $\mathcal{A}$ on $\mathcal{P}$ and also reports that $\mathcal{A}$ is insecure if a taint propagates to the watchdog timer in any possible state, assurance of security from $\mathcal{T}_{\mathcal{S}}$ means that it is impossible for tainted data to propagate through a state element and influence the execution of $\mathcal{A}_{\mathcal{I}}$.

Case 2 – taint propagation through memory: For taintedness to influence $\mathcal{A}_{\mathcal{I}}$ through the memory, a tainted computational task $\mathcal{A}_{\mathcal{J}}$ must write to some memory location $\mathcal{M}$ outside its tainted memory partition, and $\mathcal{A}_{\mathcal{I}}$ must read from that memory location while it is executing on $\mathcal{P}$. However, in any case where $\mathcal{T}_{\mathcal{S}}$ identifies that $\mathcal{A}_{\mathcal{J}}$ could write outside of its memory partition, $\mathcal{A}$ is modified such that masking instructions are inserted to ensure that $\mathcal{A}_{\mathcal{J}}$ can only write inside its own memory partition. Furthermore, $\mathcal{T}_{\mathcal{S}}$ checks all possible execution states of $\mathcal{A}$ on $\mathcal{P}$ and reports that $\mathcal{A}$ is insecure if a tainted write is performed to untainted memory or a read is performed from tainted memory by any untainted computational task. Therefore, assurance of security from $\mathcal{T}_{\mathcal{S}}$ means that it is impossible for tainted data to propagate through memory and influence the execution of $\mathcal{A}_{\mathcal{I}}$.

Case 3 – taint propagation through a port: For taintedness to propagate to an output through a port, either some $\mathcal{A}_{\mathcal{I}}$ must read from a tainted port or some $\mathcal{A}_{\mathcal{J}}$ must write directly to an untainted port. Both cases are reported as insecure by $\mathcal{T}_{\mathcal{S}}$ as it evaluates all possible execution states of $\mathcal{A}$. Therefore, assurance of security from $\mathcal{T}_{\mathcal{S}}$ means that it is impossible for tainted data to influence the execution of $\mathcal{A}_{\mathcal{I}}$ or propagate to an untainted output from a port. ■

7.4 Results

We evaluate the information flow security of each benchmark running as a tainted computational task on the system (ports it uses are labeled tainted). System code is an untainted task consisting of the instructions needed to restart the benchmark after each execution.

7.4.1 Information Flow Violations

Application-specific gate-level information flow tracking reports all possible information flow violations for an application. Table 7.1 shows which of the unmodified benchmarks violated the sufficient conditions described in Section 7.3.1. Seven benchmarks do not violate any of the conditions. Effectively, our analysis shows that these benchmarks cannot violate our information flow security policy on this processor. However, six benchmarks violate sufficient conditions 1 and 2.[4] These benchmarks require the techniques described in Section 7.3.2 to guarantee information flow security. After performing software modifications identified by our toolflow, all condition violations are eliminated.[5] Thus, symbolic gate-level information flow tracking in conjunction with software modification is able to guarantee information flow security for these applications on a commodity processor without hardware-based information flow control mechanisms.

Table 7.1: Benchmarks that violate sufficient conditions 1 and 2 for information flow security (see Section 7.3.1) before and after modification.

Benchmark	No Mod		Modified	
	C1	C2	C1	C2
binSearch	✓	✓	-	-
div	✓	✓	-	-
inSort	✓	✓	-	-
intAVG	✓	✓	-	-
intFilt	-	-	-	-
mult	-	-	-	-
rle	-	-	-	-
tHold	✓	✓	-	-
tea8	-	-	-	-
FFT	-	-	-	-
Viterbi	✓	✓	-	-
ConvEn	-	-	-	-
autocorr	-	-	-	-

Table 7.2: Performance overhead (%) for watchdog timer reset and memory address masking applied with and without application analysis.

Benchmark	W/o App	With App
binSearch	34.63	34.63
div	33.16	33.16
inSort	37.92	10.00
intAVG	45.56	11.90
intFilt	19.58	0
mult	150.9	0
rle	45.61	0
tHold	106.2	106.2
tea8	93.89	0
FFT	17.63	0
Viterbi	1.029	1.029
ConvEn	19.69	0
autocorr	42.15	0

[4] None of our benchmarks violate sufficient conditions 3, 4, or 5; however, this is not surprising for well-written code, since the conditions preclude scenarios like reading memory out of bounds or illegal port accesses.

[5] When *-logic analysis was used to verify information flow security on the six applications with information flow violations, it identified that the condition violations were not removed. This is because these applications have control dependences on an unknown, tainted input, which causes *-logic to taint the PC and make it unknown, resulting in 70% of the gates in openMSP430 becoming unknown and tainted, even those required by the software techniques to remain untainted (e.g., the watchdog timer). Therefore, a direct application of *-logic analysis would not allow the software-based techniques to be verified on commodity hardware.

7.4.2 Runtime Overheads of Software-based Gate-level Information Flow Security

Since we eliminate possible tainted information flows through software modification, guaranteeing information flow security in our approach incurs performance and energy overheads whenever an application has potential violations to eliminate. The right column of Table 7.2 (With App) shows the performance overhead of using the watchdog timer and memory masking to eliminate information flow security vulnerabilities in our benchmark applications. Since application-specific gate-level information flow tracking is able to identify and eliminate only the tainted information flows that an application is vulnerable to, applications that are not vulnerable to tainted information flows require no modifications and incur no overhead. For applications where modifications are necessary, masking is applied to store instructions that may be tainted, and the watchdog timer is used to deterministically bound the execution time of tainted computational tasks.

Since the openMSP430 watchdog has a maximum interval length of 32768 cycles, which may not be long enough to bound the longest execution time of a computational task, we evaluate a system that implements time-slicing (e.g., as an RTOS might schedule one computational task across multiple time slices). Also, since the execution time of a task may not be an even multiple of one of the available watchdog timer intervals (64, 512, 8192, and 32768 cycles), an infinite idle loop is added at the end of each benchmark to fill the remainder of the final time slice. The number and duration of time slices are selected to minimize overhead, based on the available watchdog timer intervals and the overhead of state checkpointing and recovery (context switching) for time slicing.[6] Intuitively, using fewer, longer time slices for a given task duration incurs less overhead for context switching but may incur more idling overhead in waiting for the final watchdog interval to complete. Our toolflow accounts for the overheads of context switching and scheduling the watchdog timer, along with the maximum duration of a computational task, to select the number and duration of watchdog intervals that minimize overhead while providing a deterministic bound on execution time.

Since application-specific gate-level information flow analysis indicates precisely which

[6] For openMSP430, the overhead of saving and restoring a task's state is 20 cycles, and watchdog timer initialization and reset takes 10 cycles.

computational tasks need to be protected by a watchdog timer and which store instructions need to be protected by address masking, the techniques can be applied only where necessary. On the other hand, guaranteeing information flow security for an unknown application requires masking of every store and time bounding of every tainted task using a deterministic timer, since all sufficient conditions must be satisfied to guarantee non-interference, even though they may not be necessary for a particular application (Section 7.3.1). Without the ability to identify all possible tainted information flows for all possible executions of an application on a commodity processor using input-independent gate-level information flow tracking, an "always-on" approach for information flow control would be required.

The left column of Table 7.2 (W/o App) shows the performance overhead of using masking for all stores and time bounding for all tainted tasks, representing a case where application analysis is not available and all sufficient conditions must be enforced. In this case, performance overhead is 3.3× higher than in the case where application analysis is able to target only the possible insecure information flows for an application. Even considering only the applications that have possible information flow security violations, applying software-based techniques only where necessary reduces performance overhead by 24%. Overall, application-specific information flow analysis can minimize the overhead of providing information flow security guarantees on a commodity processor using software-based techniques.

7.4.3 Information Flow Secure Scheduling: A System-level Use Case

In this section, we show that we can use the techniques developed in this work to guarantee information flow security at the system level; we focus on an IoT system that performs scheduling between multiple tasks. Specifically, we show that without any modifications to the processor, we can guarantee that (1) there are no insecure information flows across scheduled tasks, and (2) no task can affect the scheduling performed by the system software. In order to demonstrate these properties, we construct an IoT system in which FreeRTOS [96] performs task scheduling between two tasks – `div` and `binSearch` – where `binSearch` is an untrusted task (its input and output ports are marked as untrusted), and FreeRTOS and `div` are both trusted.

The control flow of `binSearch` depends on an untrusted input value. Thus, in the baseline case, after `binSearch` is scheduled on the processor, the processor's control flow becomes tainted. Among the consequences of this tainting are that (1) the trusted task `div` becomes untrusted the next time it is scheduled, and (2) the scheduling of FreeRTOS itself is compromised, since it too becomes untrusted as a result of the tainted task.

To provide information flow security for this system, we modify the system's application, consisting of FreeRTOS and the two computational tasks. 330 store instructions in `binSearch` are identified as potential security violations, and our toolflow applies memory masking to these instructions. Also, our toolflow invokes the watchdog timer mechanism around the untrusted task. This modification is performed in FreeRTOS system code. The value of the reset interrupt vector is set to a location in the middle of FreeRTOS's scheduler interrupt. On a watchdog-invoked reset, scheduling is performed as usual with the exception that the watchdog timer is also reset with the scheduling timer prior to restoring the context of the next task from its own stack. After modification, application-specific information flow tracking verifies that the application runs successfully without any tainting of the trusted task or the RTOS.

We measure the performance overhead of our modification using input-based gate-level simulations; runtime is measured from when the first task is scheduled to when both tasks have completed. The total performance overhead of adding the watchdog timer reset and memory masking is only 0.83%. The overhead is low since only `binSearch` requires memory masking and the modifications required to add the watchdog timer to FreeRTOS's system code are small (e.g., FreeRTOS already requires context saving and restoring).

The above example shows that we can guarantee gate-level information flow security with low-overhead software modifications for an application built on a commodity RTOS. More broadly, this shows that our techniques are applicable at the system-level and can be used to verify complex and system-level security properties.

Chapter 8

Conclusion and Discussion

For many emerging applications such as the internet of things and sensor networks, new challenges in design have shown up in terms of power, area, cost and security. Power and energy, for example, have become critical design constraings that not only affect the lifetime of an ultra-low-power (ULP) system, but also its size and weight. To this end this research has proposed a novel application specific hardware-software co-analysis technique that is based on symbolic simulation to capture the behavior of an application on a processor for all possible inputs to that application.

This novel hardware-software co-analysis technique has been shown to optimize a processor-application pair for power, area, cost and security. Specifically, using this technique we showed that an application specific V_{min} can be determined by exploiting available dynamic timing slack (DTS), saving power at no performance cost. We then showed that using the technique module-oblivious power domains can be designed that save double the leakage energy compared to traditional module-based power domains. The technique has been shown to also better estimate the peak power and energy of a processor application pair enabling a smaller and cheaper overall system. We also showed that application of the hardware-software co-analysis technique can enable automatic generation of application-specific 'bespoke' processors drastically reducing the design costs of application-specific designs. Finally, augmenting the hardware-software co-analysis methdology with taint information we showed that we can track information flow for an application on a processor at the gate-level enabling secure software on legacy ULP systems.

The methodology proposed in this research is novel and can enable further research in this direction such as designing application specific instruction set architectures from a generic instruction set architecture. This work can also enable construction of design-aware compilers which can use the information from the hardware-software co-analysis technique to choose better instructions for a certain optimization metric. The core technique of hardware-software co-analysis can be further improved by adding ‘hints’ from the software level abstraction during gate-level simulation leading to faster simulation times and less conservative activity information.

References

[1] 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.

[2] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, Jan 2005.

[3] Battery energy. `http://www.allaboutbatteries.com/Battery-Energy.html`, 2015.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] Isidor Buchmann. The Secrets of Battery Runtime. *Battery University*, 2016.

[13] Kjartan Furset and Peter Hoffman. High pulse drain impact on CR2032 coin cell battery capacity. *Nordic Semiconductor and Energizer*, 2011.

[14] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[15] Randal E Bryant. Symbolic Simulation – Techniques and Applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 517–521. ACM, 1991.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Randal E. Bryant. Symbolic simulation – techniques and applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 517–521, 1990.

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

[23] 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.

[24] International Technology Roadmap for Semiconductors 2.0 2015 Edition Executive Report. http://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_i

[25] A Hakan Baba and Subhasish Mitra. Testing for transistor aging. In *VLSI Test Symposium, 2009. VTS'09. 27th IEEE*, pages 215–220. IEEE, 2009.

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

[27] H. C. Chen and D. H. C. Du. Path sensitization in critical path problem. In *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pages 208–211, Nov 1991.

[28] Jing-Jia Liou, A. Krstic, L. C. Wang, and Kwang-Ting Cheng. False-path-aware statistical timing analysis and efficient path selection for delay testing and timing validation. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 566–569, 2002.

[29] P. C. McGeer and R. K. Brayton. Efficient algorithms for computing the longest viable path in a combinational network. In *Design Automation, 1989. 26th Conference on*, pages 561–567, June 1989.

[30] Janak Patel. Cmos process variations: A critical operation point hypothesis. `www.stanford.edu/class/ee380/Abstracts/080402-jhpatel.pdf`, 2008.

[31] Andrew B Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 825–831. IEEE, 2010.

[32] Hari Cherupalli, Rakesh Kumar, and John Sartori. Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems. In *International Symposium on Computer Architecture (ISCA), 2016 43rd ACM/IEEE*, 2016.

[33] John Sartori and Rakesh Kumar. Compiling for energy efficiency on timing speculative processors. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1297–1304. IEEE, 2012.

[34] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18. IEEE, 2003.

[35] Shidhartha Das, Carlos Tokunaga, Sanjay Pant, Wei-Hsiang Ma, Sudherssen Kalaiselvan, Kevin Lai, David M Bull, and David T Blaauw. RazorII: In situ error detection and correction for PVT and SER tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):32–48, 2009.

[36] Matthew Fojtik, David Fick, Yejoong Kim, Nathaniel Pinckney, David Money Harris, David Blaauw, and Dennis Sylvester. Bubble razor: Eliminating timing margins in an arm cortex-m3 processor in 45 nm cmos using architecturally independent error detection and correction. *Solid-State Circuits, IEEE Journal of*, 48(1):66–81, 2013.

[37] Giang Hoang, Robby Bruce Findler, and Russ Joseph. Exploring circuit timing-aware language and compilation. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 345–356, 2011.

[38] Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay, and Andreas Burg. Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 381–386, San Jose, CA, USA, 2015. EDA Consortium.

[39] D. Bull, S. Das, K. Shivashankar, G.S. Dasika, K. Flautner, and D. Blaauw. A Power-Efficient 32 bit ARM Processor Using Timing-Error Detection and Correction for Transient-Error Tolerance and Adaptation to PVT Variation. *Solid-State Circuits, IEEE Journal of*, 46(1):18–31, Jan 2011.

[40] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8. USENIX Association, 2010.

[41] Padmanabhan Pillai and Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 89–102. ACM, 2001.

[42] Vishnu Swaminathan and Krishnendu Chakrabarty. Real-time task scheduling for energy-aware embedded systems. *Journal of the Franklin Institute*, 338(6):729–750, 2001.

[43] C Mani Krishna and Yann-Hang Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *2013 IEEE 19th Real-Time*

and Embedded Technology and Applications Symposium (RTAS), pages 156–156. IEEE Computer Society, 2000.

[44] Andreas Weissel and Frank Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246. ACM, 2002.

[45] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81. ACM, 1998.

[46] Charles R Lefurgy, Alan J Drake, Michael S Floyd, Malcolm S Allen-Ware, Bishop Brock, Jose A Tierno, and John B Carter. Active management of timing guard-band to save energy in POWER7. In *proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–11. ACM, 2011.

[47] James Tschanz, Keith Bowman, Steve Walstra, Marty Agostinelli, Tanay Karnik, and Vivek De. Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance. In *VLSI Circuits, 2009 Symposium on*, pages 112–113. IEEE, 2009.

[48] Xiaobin Yuan, Pawel Owczarczyk, Alan J Drake, Marshall D Tiner, David T Hui, John P Pennings, Francesco A Campisano, Richard L Willaman, Leana M Cropp, and Rudolph D Dussault. Design Considerations for Reconfigurable Delay Circuit to Emulate System Critical Paths. 2014.

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

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

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

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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. IEEE Press, 2013.

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

[74] 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.

[75] 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.

[76] 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.

[77] B.H. Calhoun, S. Khanna, Yanqing Zhang, J. Ryan, and B. Otis. System design principles combining sub-threshold circuit and architectures with energy scavenging mechanisms. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 269–272, May 2010.

[78] Silicon Laboratories. Energy Harvesting Reference Design User's Guide. 2011.

[79] *EEMBC, Embedded Microprocessor Benchmark Consortium.*

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

[81] Seetharam Narasimhan, Hillel J Chiel, and Swarup Bhunia. Ultra-low-power and robust digital-signal-processing hardware for implantable neural interface microsystems. *IEEE transactions on biomedical circuits and systems*, 5(2):169–178, 2011.

[82] Paul Gerrish, Erik Herrmann, Larry Tyler, and Kevin Walsh. Challenges and constraints in designing implantable medical ics. *IEEE Transactions on Device and Materials Reliability*, 5(3):435–444, 2005.

[83] Kris Myny, Steve Smout, Maarten Rockelé, Ajay Bhoolokam, Tung Huei Ke, Soeren Steudel, Brian Cobb, Aashini Gulati, Francisco Gonzalez Rodriguez, Koji Obata, et al. A thin-film microprocessor with inkjet print-programmable memory. *Scientific reports*, 4:7398, 2014.

[84] Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: system support for long-running computation on rfid-scale devices. *Acm Sigplan Notices*, 47(4):159–170, 2012.

[85] 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.

[86] K. Myny, E. van Veenendaal, G. H. Gelinck, J. Genoe, W. Dehaene, and P. Heremans. An 8b organic microprocessor on plastic foil. In *2011 IEEE International Solid-State Circuits Conference*, pages 322–324, Feb 2011.

[87] BK Charlotte Kjellander, Wiljan TT Smaal, Kris Myny, Jan Genoe, Wim Dehaene, Paul Heremans, and Gerwin H Gelinck. Optimized circuit design for flexible 8-bit rfid transponders with active layer of ink-jet printed small molecule semiconductors. *Organic Electronics*, 14(3):768–774, 2013.

[88] Tsuyoshi Hamada, Khaled Benkrid, Keigo Nitadori, and Makoto Taiji. A comparative study on ASIC, FPGAs, GPUs and general purpose processors in the O (Nˆ 2) gravitational N-body simulation. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 447–452. IEEE, 2009.

[89] Tensilica Customizable Processors. http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[90] Sönke Holthusen, Sophie Quinton, Ina Schaefer, Johannes Schlatow, and Martin Wegner. Using multi-viewpoint contracts for negotiation of embedded software updates. *arXiv preprint arXiv:1606.00504*, 2016.

[91] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*, pages 94–98, Aug 2008.

[92] Steven Cherry. Hacking Pacemakers. `http://spectrum.ieee.org/podcast/biomedical/devices/hacking-pacemakers/`.

[93] Wikipedia. Bare Machine, Wikipedia. `http://en.wikipedia.org/wiki/Bare_machine`.

[94] Texas Instruments. StarterWare. `http://processors.wiki.ti.com/index.php/StarterWare`.

[95] ARM. Building BareMetal ARM systems with GNU. http://www.state-machine.com/arm/Building_bare-metal_ARM_with_GNU.pdf.

[96] The FreeRTOS website. http://www.freertos.org/.

[97] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[98] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[99] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[100] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 133–147, New York, NY, USA, 2005. ACM.

[101] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[102] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[103] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual*

International Symposium on Computer Architecture, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[104] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *2008 International Symposium on Computer Architecture*, pages 377–388, June 2008.

[105] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184, Feb 2008.

[106] H. Chen, X. Wu, L. Yuan, B. Zang, P. c. Yew, and F. T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *2008 International Symposium on Computer Architecture*, pages 401–412, June 2008.

[107] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. *SIGPLAN Not.*, 44(3):109–120, March 2009.

[108] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM.

[109] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 493–504, Dec 2009.

[110] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM.

[111] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 97–112, New York, NY, USA, 2014. ACM.

[112] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015. ACM.

[113] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 555–568, New York, NY, USA, 2017. ACM.