© 2017 Henry John Duwe III
DEPENDABLE DESIGN FOR LOW-COST ULTRA-LOW-POWER PROCESSORS

BY

HENRY JOHN DUWE III

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Associate Professor Rakesh Kumar, Chair
Professor Deming Chen
Associate Professor Lara Dolecek, University of California, Los Angeles
Associate Professor Nam Sung Kim
Abstract

Emerging applications in the Internet of Things (IoT) domain, such as wearables, implantables, smart tags, and wireless sensor networks put severe power, cost, reliability, and security constraints on hardware system design. This dissertation focuses on the architecture and design of dependable ultra-low power computing systems. Specifically, it proposes architecture and design techniques that exploit the unique application and usage characteristics of future computing systems to deliver low power, while meeting the reliability and security constraints of these systems. First, this dissertation considers the challenge of achieving both low power and high reliability in SRAM memories. It proposes both an architectural technique to reduce the overheads of error correction and a technique that uses the nature of error correcting codes to allow lower voltage operation without sacrificing reliability. Next, this dissertation considers low power and low cost. By leveraging the fact that many IoT systems are embedded in nature and will run the same application for their entire lifetime, fine-grained usage characteristics of the hardware-software system can be determined at design time. This dissertation presents a novel hardware-software co-analysis based on symbolic simulation that can determine the possible states of the processor throughout any execution of a specific application. This enables power-gating where more gates are turned off for longer, bespoke processors customized to specific applications, and stricter determination of peak power bounds. Finally, this dissertation considers achieving secure IoT systems at low cost and power overhead. By leveraging the hardware-software co-analysis, this dissertation shows that gate-level information flow security guarantees can be provided without hardware overheads.
To my grandmother, Margarete Caroline Wolf Hopkins
Acknowledgments

I would like to thank my advisor, Professor Rakesh Kumar, for his guidance throughout my graduate career, keeping me focused on the important aspects of my work while helping me develop as a well-rounded academic. I am also indebted to Professor John Sartori for his mentorship and feedback throughout this work. I would like to thank Professor Janak Patel and my committee for their helpful comments, insights, and suggestions. Their intellectual rigor was greatly appreciated. I owe a lot to my comrades with keyboards who worked tirelessly alongside me, supporting me both professionally and personally, especially Hari Cherupalli, Xun (Stevo) Jian, Matt Tomei, Weidong Ye, Daniel Petrisko, and Matt Sinclair. I would like to recognize all the logistical assistance I received from the Coordinated Science Laboratory staff, especially Carol Wisniewski; I couldn’t have managed this work without their support. Last, I must express my utmost gratitude to my family for their unconditional love, support, and advice throughout my life—I could not have made it without you: Pa, Ma, Margie, Hazel, EJ, Bernie, Brad, Gretchen, and GH. A special thanks to my wife Elise for being my partner in crime, helping me to steal time to spend with our daughter, Adelaide.
# Table of Contents

Chapter 1  Introduction .................................................. 1

Chapter 2  Correction Prediction: Reducing Error Correction Latency for On-Chip Memories ........................................ 5
  2.1 Introduction ......................................................... 5
  2.2 Background and Related Work .................................... 7
  2.3 Motivation .......................................................... 10
  2.4 Correction Prediction for L1 Caches ............................. 12
  2.5 Implementation, Coverage, and Overheads ..................... 18
  2.6 Methodology ....................................................... 24
  2.7 Experimental Results .............................................. 29
  2.8 Future Work and Discussions .................................... 35
  2.9 Summary .......................................................... 37

Chapter 3  Rescuing Uncorrectable Fault Patterns in On-Chip Memories through Error Pattern Transformation ....................... 39
  3.1 Introduction ......................................................... 39
  3.2 Motivation .......................................................... 42
  3.3 Error Pattern Transformation .................................... 44
  3.4 Methodology ....................................................... 53
  3.5 Results .............................................................. 58
  3.6 Related Work ...................................................... 66
  3.7 Summary .......................................................... 69

Chapter 4  A Scalable Approach to Symbolic Hardware-Software Co-Analysis ......................................................... 70
  4.1 Introduction ......................................................... 70
  4.2 Background and Motivating Insight ............................... 73
  4.3 Scalable Symbolic Co-Analysis ................................... 75
  4.4 Methodology ....................................................... 79
  4.5 Results .............................................................. 80
  4.6 Generality and Limitations ....................................... 85
  4.7 Related Work ...................................................... 87
  4.8 Summary .......................................................... 88
Chapter 1

Introduction

The next two decades will see an unprecedented proliferation of computing—not only will each person carry a mobile phone, but each person will live in a home with hundreds of devices, will be wearing several devices, and may even have implanted devices that are all interconnected. Such future usage of computing, broadly described as the Internet of Things (IoT) or Internet of Everything imposes several technical challenges on hardware systems (see Figure 1.1). First, since many IoT devices are likely to be run off of batteries or energy harvesters and may be implanted within people’s bodies, there is an extreme downward pressure in terms of power. Second, since there will be so many of these devices (over 30 billion devices by 2020 [1]) and so many unique applications (over 25 million applications [1]), the cost of each device and application must be low. Third, since these applications are fundamentally physically facing, there is an upward pressure on these devices to be reliable and secure—serious challenges, considering each device is likely connected to the broader internet. Addressing these challenges becomes difficult since approaches to address any one of these challenges must not exacerbate the others.

Traditional techniques for addressing the power challenge are constrained by reliability and cost concerns. For example, memory components of ultra-low-power systems can limit the power of future low-power processors. Historically, one of the most effective knobs for reducing power consumed by hardware has been voltage scaling. Unfortunately, as voltage is pushed to ultra-low levels, the device variability of current and future technology nodes

---

The author has been very fortunate to be a part of several collaborative efforts. Specifically, the author would like to acknowledge that the symbolic simulation based co-analysis work was co-led by the author and Hari Cherupalli. The author would also like to acknowledge the contributions that Weidong Ye, Xun Jian, and Daniel Petrisko made to various aspects of the work presented in this dissertation.
Figure 1.1: The Internet of Things (IoT) imposes several severe technical challenges. There is a large downward pressure in power and cost and significant upward pressures in reliability and security.

leads to high fault rates in SRAMs. As a result of this reliability concern, the power of memories dominates chip power or all of the chip, including logic, must be run a higher voltage determined by the SRAM memories. To enable further power advantages from voltage scaling while still maintaining reliability for future power constrained computing systems, this dissertation explores providing low-cost techniques to correct high SRAM fault rates.

Another traditional approach to producing ultra-low-power devices is using application-specific integrated circuits (ASICs). ASICs are custom designed to efficiently perform the unique task(s) required by an application. However, ASICs incur high costs for application development and maintenance (e.g., design and verification effort, debuggability, compiler toolchain support, and lack of in-field updates). Therefore, many future IoT applications will be powered by general-purpose embedded microprocessors and microcontrollers [2, 3, 4] rather than ASICs. This dissertation explores the following question: Are there unique opportunities for power reduction of emerging IoT applications that exploit the embedded nature of these applications given that these applications are often driven by ultra-low-power microprocessors and microcontrollers?

In answer to this question, we propose a novel co-analysis of both the soft-
ware run on the programmable processor and the gate-level netlist of that processor. This co-analysis, based on a form of symbolic simulation, can provide information about the state of each gate of the processor throughout any possible execution of the application. With this information, one can determine which gates are used (i.e., toggled) at which points during the execution of the application, enabling a plethora of optimizations. One use of such information is to define and control module-agnostic power-gating domains that allow more gates to be turned off for longer even with the same number of power domains. A more aggressive optimization all together removes the unused gates from the design, saving power. Designing such a bespoke processor also provides significant area reduction, reducing the cost of a chip. Another cost-saving approach is to use the detailed activity information from our hardware-software co-analysis to determine a tight bound on the peak power and energy that a microprocessor can use during the execution of a specific application. By accurately determining the required power and energy an application uses, smaller capacity (and thus cheaper) harvesters and batteries can be used. Finally, the hardware-software co-analysis can allow the design and verification of software-based techniques that guarantee information flow security at the gate level (i.e., the finest digital granularity). These uses represent a series of tools that provide system designers the ability to develop new applications that are ultra-low-power and cost while maintaining reliability and security.

Overall, this dissertation focuses on the architecture and design of dependable ultra-low power computing systems. Specifically, it proposes architecture and design techniques that exploit the unique application and usage characteristics of future computing systems to deliver low power, while meeting the reliability and security constraints of these systems. The remainder of this dissertation is organized as follows. Chapter 2 presents an architectural technique to tolerate the high latency overheads of error correction for high fault rate on-chip memories. Chapter 3 presents an approach that uses the characteristics of error correction codes to reduce power further. Chapter 4 describes the core underlying scalable symbolic simulation methodology that enables dependable application-specific optimizations. Chapter 5 argues for defining power-gating domains at a fine-granularity across modules. Chapter 6 describes an automated toolflow to reduce the area and power of a processor by removing gates unused by a specific application. Chapter 7
describes how the hardware-software co-analysis can be used to guarantee the peak power and energy of a processor. Chapter 8 proposes an approach to provide gate-level information flow security entirely in software. Last, Chapter 9 summarizes the work.
Chapter 2

Correction Prediction: Reducing Error
Correction Latency for On-Chip Memories

The reliability of on-chip memories (e.g., caches) determines their minimum operating voltage ($V_{\text{min}}$) and, therefore, the power these memories consume. A strong error correction mechanism can be used to tolerate the increasing memory cell failure rate as supply voltage is reduced. However, strong error correction often incurs a high latency relative to the on-chip memory access time. In this chapter, we propose correction prediction where a fast mechanism predicts the result of strong error correction to hide the long latency of correction. Subsequent pipeline stages execute using the predicted values while the long latency strong error correction attempts to verify the correctness of the predicted values in parallel. We present a simple correction prediction implementation, CP, which uses a fast, but weak error correction mechanism as the correction predictor. Our evaluations for a 32KB four-way set associative SRAM L1 cache show that the proposed implementation, CP, reduces the average cache access latency by 38%-52% compared to using a strong error correction scheme alone. This reduces the energy of a two-issue in-order core by 16%-21%.

2.1 Introduction

One simple, yet effective, technique to reduce the power of on-chip memories (e.g., caches) is voltage scaling [5, 6, 7, 8, 9]. Reducing the supply voltage results in significant reductions in static and dynamic power [8]. One major challenge of scaling the voltage of on-chip memories is maintaining the desired reliability. Process variations can cause a rapidly increasing fraction of memory cells to become faulty as the supply voltage decreases [8, 10].

A flurry of recent work has been devoted to providing the desired cache reliability at low supply voltages [8, 9, 11, 12, 13]. Some propose using
larger, and therefore stronger, memory cells (e.g., 8T and 10T SRAM cells or cells with up-sized transistors) to prevent errors from occurring in the first place. Unfortunately, these methods incur a high static overhead even for nominal voltage operation (see Section 2.6.1). As a result, many have instead proposed using error correction to correct the wrong values in memory cells as the cells become faulty due to voltage scaling. A large number of error correction techniques have been proposed, spanning from the use of error correction codes [9, 11, 13] to data remapping [8, 12].

However, error correction inevitably incurs a latency overhead, which may be significant relative to the cache access time for error correction strong enough to provide the desired reliability (see Section 4.2). This increase to cache access time due to strong error correction may lead to a significant degradation in performance and energy (see Section 2.7) from either an increased clock cycle time [14] or increased pipeline depth. An alternative supported by some processors is to speculatively execute on the instruction or data accessed from the cache prior to performing error detection [15]. A detected error corresponds to a mis-speculation, which causes the appropriate instructions to be squashed and the pipeline to be stalled for correction. While this technique suffices for scenarios where the bit-failure probability is low, it incurs a high performance overhead for scenarios where the bit-failure probability is high (e.g., when the supply voltage is low) due to rampant mis-speculation leading to frequent squashes and stalls (see Section 2.4.1).

We propose a novel scheme for hiding the latency overhead of a strong error correction scheme used to ensure reliability. Our scheme, correction prediction, uses a fast mechanism to predict the results of strong error correction. Subsequent pipeline stages execute using the predicted values. In parallel, the long latency strong error correction attempts to verify the correctness of the predicted values. On a mis-prediction, i.e., when the value produced by the correction predictor is not the same as the result of the strong error correction, speculative instructions are squashed and the pipeline is re-started. By allowing the logic core to execute on the predicted data or instructions, one can effectively hide the latency of the slow, strong error correction, even at very low supply voltages where cell failure is prevalent. In the context of hard faults in voltage scaled SRAM L1 caches, we propose implementing the correction predictor using a fast, but weak error correction mechanism that produces the same result as strong error correction mechanisms for most but
not all words. Our implementation, CP, is based on a Correction Prediction Table (CPT—details in Section 2.5) that can correctly predict over 90% of cache word corrections.

We make the following contributions:

- We propose correction prediction, a scheme to reduce the latency of strong error correction by predicting its output. Long latency strong error correction verifies the correctness of prediction even as execution continues with the predicted value.

- We present CP, a simple implementation of correction prediction where a fast, weak correction precedes strong error correction and allows CP to limit the mis-prediction rate to $<0.1\%$. CP adds $<10\%$ area overhead and $<2.5\%$ worst-case latency to a cache with strong error correction.

- We evaluate CP applied to three recently proposed strong error correction schemes—Hi-ECC [16], VS-ECC [11], and Bit-Fix [8]. Compared to using the strong error correction technique alone, CP reduces L1 cache access latency by 38\%, 38\%, and 52\%, respectively. For a two-issue in-order core, this corresponds to a processor-wide energy savings of 16\%, 17\%, and 21\%.

2.2 Background and Related Work

In this chapter, we apply correction prediction to SRAM-based L1 caches. Below we discuss the effect of voltage scaling on SRAM cache reliability and prior approaches to provide reliable cache operation.

2.2.1 SRAM Reliability at Low Voltages

Some SRAM cells in a cache are weaker than other cells in the cache due to process variations. Although practically every cell in a cache (e.g., 99.999\% plus) is functional at a high supply voltage, more and more of these weaker cells become faulty as the supply voltage is reduced. A faulty cell can experience both read failures, where the wrong value is returned or the stored value is toggled unintentionally, and write failures, where the value in the cell cannot be toggled [8]. Since these faulty cells are due to permanent defects
(e.g., dopant variation), they can be located using a number of built-in self-test (BIST) routines [8, 11, 12]. Some faults at low voltages are due to soft errors [17] that cannot be detected by BIST routines. However, the fraction of such faults at low voltages is minuscule (by over 5 orders of magnitude at 650mV [9]).

Figure 2.1 shows the average fraction of the cells in a 65nm cache that are faulty as a function of the supply voltage. The calculation is based on the SRAM failure probability reported in [10] and assumes that the faulty cells are distributed randomly across the cache, which is consistent with the assumption made in prior works [8, 9, 11]. We also calculated the fraction of 32-bit words that are faulty; a word is faulty if it contains one or more faulty bits. The results in Figure 2.1 show that nearly 30% of all words require error correction when the supply voltage is scaled beyond 650mV, motivating the need for efficient error correction algorithms.

2.2.2 Error Resilience Techniques in Caches

One technique to improve reliability is using larger transistors to implement the SRAM cells [10]. Another technique is to use a more fault tolerant SRAM implementation, such as an 8T or a 10T SRAM cell [10]. The downside of these techniques is that they incur a significant area and power overhead.
even when the processor is operating at high supply voltage.

There have been several recent attempts at using strong error correcting codes to implement a cache that operates reliably at low supply voltages while incurring a low overhead at a high supply voltage. For example, FLAIR [13] uses a combination of SECDED (single error correction double error detection) codes and dual modular redundancy to correct errors in a cache line. VS-ECC [11] proposes using a combination of SECDED and 4EC5ED (four error correct five error detect). MS-ECC [9] proposes trading off storage-overhead for decode latency by using Orthogonal Latin Square Codes (OLSC) for multi-bit error correction. The downside of these techniques is their high energy overhead at low voltages due to the high performance cost of detection and correction (Section 2.7).

Finally, several works observe that since the location of the faulty cells can be predetermined via offline testing (e.g., BIST), one can remap the value of faulty cells elsewhere in the cache. PADded Cache [18], for example, uses a fast, programmable address decoder to remap cache lines into other sets and disables faulty physical lines. Unfortunately, for low supply voltage operation, most cache lines would need to be disabled (e.g., >99% of cache lines at 650mV). Bit-Fix [8] proposes a simple remapping policy that uses dedicated bits per cache line to record the bad bits in each cache line and remap their values to a different cache line. Archipelago [12] proposes a more sophisticated remapping policy that uses a global fault map table to perform remapping with greater flexibility. The primary downside of these techniques is the unavoidable latency increase for every cache access due to data correction after data array access or map look-up before the data access. This latency increase may have significant performance and energy impact (Section 2.7).

2.2.3 Tolerating Error Detection and Correction Latency

One simple approach to account for the additional delay in the instruction-fetch and data-load stages due to error correction without stalling the processor pipeline is to slow down the overall core frequency; however, this can lead to a significant performance degradation (Section 2.7) since the error correction latency is often a significant fraction of the cache access latency.
Instead of slowing down the core frequency, Bonnoit et al. [14] propose using additional pipeline stages to handle the error correction latency. However, using pipeline deepening to hide correction latency can also lead to a degradation in performance and energy efficiency (Section 2.7) because branch and data hazards become more expensive. Also, load-dependent instructions are stalled more frequently and for more cycles. Furthermore, since only low supply voltage operation requires error correction, the added pipeline stages result in unwanted overhead for high voltage operation.

Bonnoit et al. [14] also propose avoiding the additional pipeline stages by decoupling error detection from correction. They observe that error detection typically incurs a shorter latency than error correction. Therefore, they propose reducing the clock frequency to accommodate only the error detection latency, and then stalling the pipeline when errors are detected to wait for the error correction. However, this technique may result in a significant reduction in clock frequency when error detection latency is still a large fraction of cache access latency. In addition, this technique leads to frequent stalling when the fraction of cache accesses with errors is high, which limits its effectiveness for aggressive voltage scaling.

Some processors [15] attempt to hide the latency of error detection by using speculation, whereby the word retrieved from the cache is sent directly to the subsequent pipeline stages, prior to performing any error detection; meanwhile, error detection takes place in the clock cycles following the cache access. If errors are detected, the pipeline is flushed, an exception handler performs the correction, and execution restarts from the erring instruction. However, this technique is also ineffective at low voltages, where flushing is frequent due to the large fraction of cache accesses with errors (see Figure 2.1).

2.3 Motivation

Error correction can be expensive when the number of errors that need to be corrected is large (as may be the case for low $V_{\text{min}}$ L1 caches). Consider, for example, 5-bit BCH-based error correction at 650mV for the SRAM failure rates in Figure 2.1. BCH-based correction has been used for strong error correction in past works such as VS-ECC [11]. We calculate the decode latency
of the BCH code for the three scenarios (no error, one error, multi-bit errors) assuming 32 data bits per codeword using the methodology in [19, 16] and using the FO4 delay for the 65nm technology reported in [20] (more modeling details in Section 2.6.1). Figure 2.2 shows the latency values normalized to the access latency of a 65nm four-way set-associative 32KB cache with 64-bit output granularity (see Section 2.6.1 for details). The figure shows that even in the most optimistic scenario (i.e., codeword is error-free), the decode latency of the BCH code is a significant fraction (50%) of the cache access latency. The decode latency is 72% of the cache access latency for single-bit errors and 647% for multi-bit errors.

Other strong error correction techniques that provide comparable reliability are expensive as well. Figure 2.2 shows the decode latency of a 7-bit error-correcting OLSC code [9], and Bit-Fix, a fault-map-based technique [8]—the reported implementations provide the same reliability as the 5-bit BCH discussed above. Results show that the decode latency of OLSC and the decoding and shifting latencies of Bit-Fix are also significant (41% and 140% of the cache access latency respectively). Moreover, while the error correction latency of the OLSC code is shorter than that of the BCH code, it comes at a significant cost in terms of storage overhead. Similarly, while the error correction latency of seven-modular redundancy is only 6% of that of the cache access, the corresponding storage overhead is 600%.

Our goal is to develop a technique that allows strong error correction to be
used for low $V_{\text{min}}$ caches without the prohibitive latency or storage overhead. Toward this goal we employ correction prediction used in conjunction with a strong error correction scheme.

2.4 Correction Prediction for L1 Caches

Correction prediction for L1 caches feeds predicted values to the pipeline while using strong error correction in parallel. When the predicted value is correct (i.e., the word consumed by the subsequent pipeline stages is the same as the output of the strong error correction), the latency of strong error correction is avoided. On a mis-prediction (i.e., the word consumed by the subsequent pipeline stages differs from the output of the strong error correction), the instructions dependent on the consumed predicted word are flushed and restarted using the output of the strong error correction. The mis-prediction penalty is the larger of the squash/restart and strong error correction penalties.

2.4.1 Key Idea

A correction predictor must be both accurate and fast. A correction predictor must have high accuracy to be effective since a large fraction of words are faulty at low voltages (e.g., over 30% at 650mV, see Figure 2.3). A high mis-prediction rate will lead to frequent squashes. A correction predictor must be fast since even an added cycle of latency may be prohibitive for L1 cache accesses.

A high accuracy correction predictor implementation predicts correctly in high likelihood scenarios. Figure 2.3 shows that for voltage-scaled SRAMs, a high likelihood scenario is where a word has zero, one, or two faults. An error correction mechanism that can correct up to two errors would correct over 99% of words at 650mV. One fast implementation of such an error correction mechanism is storing information about up to two faults in a table. Since the table cannot store enough information to correct every word and since the table itself can suffer faults, the table cannot correctly predict every access. However, we show in Section 2.5 that it can correctly predict over 90% of all accesses. By allowing the pipeline to speculatively execute using instructions
Figure 2.3: Word error distribution at 650mV. 99% of words have two or fewer errors. This suggests that a weak error correction mechanism provides sufficient prediction for most cache accesses.

or data that have been predicted by the fast error correction mechanism, our proposed error correction implementation, CP, can effectively provide error correction latency similar to that of a fast error correction mechanism with the same level of reliability as a long latency error correction mechanism.

Figure 2.4 shows the capacity overhead and the average error correction latency of the CP implementation relative to other strong error correction implementations that meet the same reliability target (i.e., implementation can correct 99.9987% of words in the cache for the word error rates shown in the legend\(^1\)). Each data point corresponds to a particular implementation of a correction technique. For the BCH code, the average correction latency is the average of the correction latency for zero errors, one error, and more than one error weighted by the frequency of occurrence of these errors (Figure 2.3). Each CP design point uses BCH as the strong error correction mechanism while the fast correction mechanism is based on fault-map and can correct up to zero, one, and two errors (corresponds to zero, one, and two Map Units—see Section 2.5). The results show that CP indeed provides latency similar to the low latency correction techniques (e.g., N-modular redundancy) at the

\(^1\)This means that each implementation has barely enough ECC resources to provide the same reliability for the voltages shown in the legend as the cache has at 1.2V.
capacity overhead of capacity-optimized techniques (e.g., BCH).

2.4.2 Microarchitectural Support

Figure 2.5 details the logical flow of an L1 cache access using CP. When the pipeline requests an L1 cache access, the cache performs the normal cache tag and data array accesses. In parallel, the cache reads the Correction Prediction Table entry (see Section 2.5) corresponding to the word being accessed. The entry indicates whether to perform correction prediction. To predict, the cache applies fast error correction and feeds the resulting predicted value to the pipeline. After speculative execution begins using the predicted value, the slow, strong error correction determines whether the predicted value contains an error. If the predicted value does contain an error, a mis-prediction has occurred and the pipeline squashes all dependent instructions. The cache
then returns the corrected value to the pipeline and the pipeline restarts the instruction that initiated the cache request with the correct value. When prediction is not performed, the cache applies strong correction to the value returned from the data array and returns the correct value to the pipeline.

To support CP, the following changes need to be made to the processor pipeline:

**Cache Support** Figure 2.6 depicts the additions the CP cache module requires beyond a traditional cache. The fast, weak correction module contains the Correction Prediction Table and associated logic (see Section 2.5). This module determines whether prediction should be triggered and also generates the predicted value. The slow, strong correction module uses the predicted value to provide sufficient error correction to meet the reliability requirement. It outputs both whether it detected an error within the predicted codeword and also returns the corrected value (i.e., the corrected data bits from the codeword) in case of an error. If the corrected value does not match the predicted value, then a mis-prediction occurred. The Pipeline Control module then indicates that a squash is required. The Pipeline Control module also stalls the pipeline when the fast, weak correction module does not predict a value and the strong error correction module must perform long-latency error correction.

We recognize that the additional logic inserted on the critical path of a cache access may result in a reduction in operating frequency. We quantify this overhead in Section 2.5.2 and study the sensitivity of benefits to this overhead in Section 2.7.3.

**Instruction Fetch** Applying CP to the instruction cache requires support for squashing each instruction dependent on a mis-predicted instruction and restarting the corrected instruction in the decode stage. Also, the predicted instructions may cause exceptions (e.g., illegal opcode or divide by zero exceptions). Such exceptions must be suppressed until strong correction completes. For the core used in our evaluations (see Section 2.6), our strong correction scheme requires at least two cycles to detect a mis-prediction, resulting in potentially erroneous instruction bits propagating to the decode, execute, and initial fetch stages. For this core, all exceptions must be suppressed until after the execute stage, which would happen anyway to maintain precise exception handling.
Correction prediction for an L1 cache access.

Figure 2.5: Correction prediction for an L1 cache access.
Figure 2.6: Modifications to L1 caches. The critical path for the common case of correction prediction is in **bold**.

**Data Memory Load** The core idea behind CP is allowing computation to continue successfully speculating during error correction. For a data memory load, this means that the predicted result must be forwarded to any dependent instructions within the pipeline. This avoids execution of dependent instructions being stalled by the additional latency of error correction. To allow continued forwarding of predicted data to dependent instructions, additional pipeline stages must be added after the data memory stage. The number of required additional stages is equal to the number of cycles it takes to perform strong correction. These additional stages are dummy stages through which instructions flow. These stages also support the forwarding of predicted values to earlier stages. Note that adding these forwarding stages to the data cache access has a significantly smaller impact on performance than adding pipeline stages for error correction (Deep Pipelining). This is because the dependent instructions are stalled waiting for the strong correction to complete in the latter case. For the core used in our evaluations (see Section 2.6), at least two additional pipeline stages are needed after the data memory stage. Mis-predictions are detected during the write-back stage. On a mis-prediction, the value generated by strong error correction is written back to the register file; all following instructions are squashed.
2.4.3 Tag Array Protection

Note that the above discussion assumes that correction is performed only on the content of the data array, not the tag array. The tag access is assumed to be robust at low voltages similar to previous related work [12, 8]. Since the tag array is significantly smaller than the data array and is less latency constrained, we assume a 10-T SRAM-based [21] implementation for the tag array to guarantee robustness at low voltages.

2.5 Implementation, Coverage, and Overheads

For correction prediction to be beneficial, the underlying prediction mechanism must only add minimal latency to a regular cache access. As such, our design goal is to limit the latency of correction prediction to the delay of a single logic gate. Many prediction mechanisms with varying prediction accuracies, latencies, and storage overheads are possible. Here we present one such correction prediction mechanism and leave a full exploration of correction prediction mechanisms as future work.

The proposed correction prediction mechanism is a fast but weak error correction mechanism that duplicates a small number of faulty bits in the cache. The number of duplicate bits is kept small so that they can be stored in a small enough table, called the Correction Prediction Table or CPT, that accessing the table is much faster than accessing the L1 cache. The CPT is accessed in parallel with a L1 cache access. The fast CPT allows the duplicate bits to be accessed and then processed before a L1 cache access completes; as such, it allows the duplicate bits to correct the faulty bits in the cache word at the cost of the delay of a single MUX, which decides what bit—a regular data bit or a duplicate bit—to output per bit position to subsequent pipeline stages.

The CPT is organized as follows. There is a CPT entry corresponding to every consecutive four words (e.g., 128 bits) in the L1 cache. Each CPT entry contains four predFlags and two Map Units (Figure 2.7). There is a predFlag for each word that allows CP to avoid bad predictions, such as when a CPT entry does not have enough Map Units to correct all faults in the corresponding words or when a Map Unit is faulty. The predFlag indicates to the cache controller whether to perform correction prediction
or to simply perform strong error correction and stall the pipeline when the word is accessed. A Map Unit has three fields—*valid*, *location*, and *value*. The valid bit indicates that the corresponding location and value fields are error-free. The location field contains the location of one of the single-bit errors within the 128 bits. The value bit contains the correct current value for the corresponding cache bit. Note that both the valid bits and the predFlags are vulnerable to errors; however, errors in the CPT only affect prediction accuracy, not reliability.

Each CPT entry is populated as follows. Only a CPT entry’s value bits are set and updated during regular cache accesses, while all other bits in the CPT are set at runtime by a built-in self-test (BIST) routine that tests the L1 cache at the target low voltage. The BIST routine first tests the two Map Units of the CPT entry. If the routine finds any faulty bits in a Map Unit, the routine sets the valid bit of the Map Unit to false. The routine then tests the 128 cache bits that correspond to the CPT entry to identify as many faulty cache bit locations as there are valid Map Units in the CPT entry. These faulty cache bit locations are then recorded in the location field of the valid Map Units. Next, for each of the four predFlags in the CPT entry, the routine sets a predFlag to true if every faulty bit in the 32-bit cache word that corresponds to the predFlag has a corresponding valid Map Unit. Finally, for each write to the cache data array (data write or cache fill), the corresponding CPT entry of written cache word must be read to update the value bits.

Figure 2.8 shows the fast correction circuitry that uses entries from the CPT to fix an error in a bit of the data word. The location field bits of the two Map Units are decoded to determine which of the 128 bits are replaced by the values stored in the Map Units. The Map Unit valid bit enables this decoding. The least significant two bits of the word offset are used.
Figure 2.8: Fast, weak error correction circuit for bit $x$ of word $i$. *M.U.* stands for Map Unit, *V.* stands for valid bit, and *L.* stands for location field. Value propagation before data array access completes is **bolded**.
Table 2.1: Prediction and mis-prediction rates.

<table>
<thead>
<tr>
<th>Rate</th>
<th>Derived Formula</th>
<th>Value at 650mV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>$1 - (p \cdot [1 - P(error)] + (1 - p) \cdot P(error))$</td>
<td>91%</td>
</tr>
<tr>
<td>Mis-prediction</td>
<td>$p \cdot P(error)$</td>
<td>0.089%</td>
</tr>
</tbody>
</table>

to determine if the Map Unit points to $Word_i$. For every bit in the word accessed from the cache ($Word_i.Bit_x$), a multiplexer is used to select between the bit read out from the cache and the values stored in the Map Units. The valid bit of a Map Unit enables the selection of its corresponding value field.

2.5.1 Detection and Correction Coverage

In order to evaluate the fast error correction mechanism we must calculate what fraction of words it attempts to predict (the prediction rate) and of those words how many are incorrectly predicted (the mis-prediction rate).

To calculate the prediction and mis-prediction rates, we first calculate the probability of observing an error in the output of fast correction. An error may exist in the output of fast correction if the total number of fault bits among the four data words that share the same CPT entry exceeds the number of correct Map Units in the entry. Note that since the Map Units themselves have the same bit error probability as the words in the L1 cache, one or more of the two correction units in a CPT entry may be faulty. To calculate the probability that an error occurs at the output of the fast correction circuitry, we observe that when the total number of faulty bits among the four data words exceed the number of non-fault Map Units by one, an error will appear when one of these four words is accessed. Therefore, the probability that an error occurs when one of these four words is accessed given that the error correction capability of the fast correction entry is exceeded by one is $1/4$. Similarly, the probability that an error will occur when one of these four words are accessed given that the error correction capability of the fast correction entry is exceeded by two is $2/4$. The following equation summarizes the probabilities of all possible scenarios that cause an error in
the output of fast correction:

\[
P(error) = \sum_{i=0}^{2} \cdot \left( \frac{2}{i} \right) \left( (1 - p)^9 \right)^{2-i} (1 - (1 - p)^9)^i \cdot [ \sum_{j=2-i+1}^{128} \left( \frac{128}{j} \right) p^j (1 - p)^{128-j} \cdot \min((j - 2 + i)/4, 1)]
\]  

(2.1)

In Equation 2.1, \( p \) is the fault probability of a single bit, \( i \) is the number of faulty Map Units in the entry and \( j \) represents the total number of bad bits in the four words with a total of 128 bits.

To determine the mis-prediction rate, \( i.e., \) the probability that the output of fast correction is faulty, but prediction is still triggered, we note that it is equal to the probability that the predFlag bit is faulty when the output of fast correction is faulty. This probability is \( p \cdot P(error) \). Prediction is not attempted when the predFlag bit is fault-free while the output of fast correction is faulty, even though it would have been beneficial to trigger prediction. The probability of this occurring is \( p \cdot [1 - P(error)] \). Prediction is also not attempted when a predFlag bit is faulty while the output of fast correction is not faulty. In this case, the pipeline is correctly stalled until strong correction completes. The probability of this occurring is \( (1 - p) \cdot P(error) \). Table 2.1 lists calculated values for our cache operating at 650mV where the bit failure rate is 0.011 (Figure 2.1). We note that we could protect the CPT with Schmidt Trigger (ST) SRAM cells or increased supply voltage. However, only 0.089% of cache accesses are mis-predicted, resulting in an insignificant performance degradation. We argue that this is a good tradeoff for not requiring an additional voltage rail or a unnecessary increase in area \( i.e., 100\% \) for the CPT.

We also note that although our specific implementation of fast correction leverages characteristics of the fault distribution \( e.g., \) uncorrelated bit errors where single errors are most common), this is not a requirement of fast error correction. For example, if faults are correlated within a word, we could increase the size of our value fields in the Map Units to improve the correct speculation rate.
2.5.2 Latency Overhead

We model the CPT for our four-way set associative 32KB cache using CACTI [22] to determine its latency. The table has a latency of 0.44ns in a 65nm technology node operating at 1.2V. As shown in Figure 2.8, the critical path of the fast correction circuit before the MUX that picks between a data bit and a duplicate bit consists of the following: a 5-to-1 binary decoder, 2 AND gates, a MUX to select between the outputs of the fast correction entries that correspond to the two ways in the set, a MUX to select between the two Map Units, and the three level of inverters required for every location to drive 32 bit slices. Using the FO4 delay of 65nm technology reported in [20], these equate to a total delay of 0.2ns. In comparison, the access latency of a four-way set-associative 32KB L1 cache (the associativity/size of the L1 caches in our evaluations in Section 2.7) is 0.68ns. Therefore, the delay of fast error correction circuitry prior to the MUX gate can be effectively hidden by the latency difference between the L1 cache and the CPT. Consequently, the MUX gate used to select between the data bit and the duplicate bit is the only additions to the critical path of the cache access. Following the methodology in [23], we estimate the MUX to be 0.5 FO4. In our evaluations, we increase the clock period of the CP cores by 0.01ns at nominal Vdd (1.2V) and 0.026ns at 650mV to account for these delays.

2.5.3 Area Overhead

The area overhead of the fast correction technique is dominated by the Correction Prediction Table. We estimated using CACTI [22] that the area of a Correction Prediction Table with two Map Units is 8.6% of the size of a four-way set associative 32KB L1 cache. Depending on the the desired level of voltage scaling, fewer Map Units may be needed. For design points with only a single Map Unit plus the four predFlag bits or with the four predFlag bits only, the area overhead due to the Correction Prediction Table normalized to cache area would be 5.1% and 1.6%, respectively.

We also estimated the area overhead of the fast correction circuit. As shown in Figure 2.8, the fast error correction circuit that corrects a single bit of the 32-bit word requires three MUXes, eight XOR gates, 12 AND gates, and four 5-to-1 binary decoders, where each requires at most four AND gates,
for a total of 40 gates. The total number of gates required by the fast error correction circuit is $40 \cdot 32 = 1240$ gates. Assuming that two such decoders are needed to keep up with the issue width of the processor, this translates to a maximum of 2480 gates, which incurs an area overhead 1.3% compared to a 32KB L1 cache.

2.5.4 Energy Overhead

We model the access energy of the CPT using CACTI [22] and determined that an access to the table results in a 4% energy increase for every cache access. We also estimated the energy consumed by the correction circuitry (Figure 2.8) assuming an activity factor of 1 and the switching energy of a 65nm transistor given in [24]. This energy is 0.2% that of the access energy of the 32KB L1 cache. Static power overheads are no worse than the area overheads discussed in Section 2.5.3.

2.6 Methodology

In this section, we describe the methodology we used to evaluate CP for a 65nm technology node. Section 2.6.1 describes the strong error correction baselines to which we apply CP. Section 2.6.2 describes the different designs we evaluated. Section 2.6.3 describes our experimental details.

2.6.1 Strong Error Correction

CP can be applied to any strong error correction technique to reduce its latency. In this chapter, we use CP in conjunction with three recently proposed strong error correction schemes—Hi-ECC [16], VS-ECC [11], and Bit-Fix [8]. These strong error correction methods vary in their area, capacity, and latency overheads.

Our Hi-ECC implementation protects every word using a BCH code that is capable of correcting five erroneous bits within a 59-bit codeword. We employ an additional parity bit to detect a sixth error within the codeword. If zero or one errors are detected, the evaluated BCH decoder only incurs the single error correction latency, but not the much longer multi-bit correction latency.
Single-bit correction is used instead of the multi-bit correction whenever applicable. The multi-bit BCH correction is used if single-bit correction fails. In our implementation, the ECC bits are stored in cache ways during low voltage operation such that half the ways store data bits, while the other half store ECC bits. The ECC cache ways store data during nominal operation as in [9]. Our VS-ECC implementation, based on VS-ECC-fixed from [11] uses seven bits for SECDED per cache word. At the same time, each cache line contains four additional 20-bit extended ECC fields to accommodate a 5EC6ED BCH code for up to four words within the line.

The slow, strong correction module in Figure 2.6 contains a BCH decoder implemented iteratively according to the modified Berlekamp-Massey algorithm presented in [11]. With the addition of a modest amount of logic, this decoder can detect errors and correct single-bit errors with a small latency relative to the latency of full 5-bit error correction [16]. Using the latency equations from [19] and the 65nm technology parameters from [20], we calculate the error detection latency for the 5EC6ED BCH code to be 0.34ns or 50% of the access latency of a four-way set-associative 32KB L1 cache [22]. Single-bit correction, as calculated from [16], takes 0.53ns or 78% of an L1 cache access. Similarly, multi-bit correction requires 4.4ns or 648% of an L1 cache access. Given the iterative BCH decoder used in [11] and Schmidt Trigger 10-T cell protection for the tag array, Hi-ECC has a total area overhead of 11.9%. VS-ECC, requiring additional static storage overhead has a 41.4% area overhead compared with our L1 cache.

The energy consumption of the BCH code used by both our Hi-ECC and VS-ECC implementations depends on the number of errors in the input codeword (i.e., zero, one, or more than one). At 650mV, an average of 11 bits are bad per 1000 bits (Figure 2.1). At this bit error rate, the energy overhead of the BCH decoding is calculated to be 0.86% that of the L1 access assuming an activity factor of 1. The energy overhead of the BCH encoder is calculated to be 0.3% that of an L1 access. Static power overheads are no worse than the previously discussed area overheads.

Our Bit-Fix implementation is adapted from [8]—we assume access at word granularity. Each access takes three cycles [8]. The decoding circuitry has fewer than 26,000 transistors [8] or roughly 1.7% of a 32KB caches data array. Our Bit-Fix implementation also requires the 4.8% area overhead for robust tag cells (10T-ST) resulting in a total area overhead of 6.5% of a 32KB L1
Figure 2.9: Impact of adding CP to different strong correction schemes at 650mV.

cache. Using an activity factor of 1, our Bit-Fix circuitry has an energy overhead of 1.2% the access energy of an L1 cache. Static power overheads are no worse than the area overheads.

Figure 2.9 presents the latency, area, and energy impact of applying CP to the above strong error correction schemes. In the worst-case, CP increases the latency of a cache access by up to 2.2% (e.g., when fast, weak correction attempts to predict a word, but strong error correction determines that the predicted word was incorrect). However, as shown in Table 2.1, most errors can be predicted by fast error correction, allowing CP to reduce the average latency of a cache access by 38% to 52% depending on the strong error correction scheme. These benefits come at an area overhead of no more than 8%, a maximum dynamic energy overhead of 4.2%, and a maximum static energy overhead of 8% at low Vdd.

Table 2.2 compares the latency, capacity, and area overheads of complete CP schemes (including overheads from both CP and the specific strong error correction scheme) to those of a strong circuit-level technique. At nominal voltage, all CP schemes have significantly smaller latency and area overheads compared to a 10T ST SRAM cell. The 10T ST SRAM [26] has significantly better reliability at low voltage than 8T [27] and 10T [28], yet it still cannot provide sufficient reliability for a 32KB cache compared with a strong error correction technique such as Bit-Fix [8].
Table 2.2: Latency, capacity, and area overheads normalized to an unprotected cache.

<table>
<thead>
<tr>
<th>Low Vdd Tolerance Technique</th>
<th>Nominal Voltage</th>
<th>Low Vdd (650mV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>+ Hi-ECC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>+ VS-ECC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>+ BitFix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10T ST Cell SRAM [25]</td>
<td>60%</td>
<td>0%</td>
</tr>
</tbody>
</table>

2.6.2 Design Points

As shown in Figure 2.9, applying CP reduces the average access latency of a reliable, low Vdd L1 cache. At the processor level, this can result in significant performance and energy benefits for those processors (e.g., in-order cores) where the latency of cache access can significantly determine performance. To quantify the processor benefits, we evaluate CP against three design points presented in Table 2.3. The first baseline, Nominal Baseline, is a seven-stage pipeline running at 2.68GHz at 1.2V. Note that cache accesses take two cycles for this baseline. The second baseline, Deep Pipe, has additional stages (for both the L1 ICache and L1 DCache) to accommodate the latency of error correction required during 650mV operation without affecting the frequency of the pipeline during nominal (1.2V) operation. For Hi-ECC and VS-ECC, error correction latency requires two cycles given the single-bit correction latency described in Section 2.6.1, while Bit-Fix requires three cycles. The frequency of the Deep Pipe baseline (and other baselines) at 650mV is determined using the operating point pairs from [8] and assuming a linear voltage-frequency scaling (this is similar to the methodology used in [29, 11]).

The third design point is Speculate on Every Access (SEA) where every cache access is speculated upon. I.e., the uncorrected value is returned to the pipeline and to the strong correction circuitry at the same time. This design point represents the natural extension of [15] using hardware error correction. This design will suffer from frequent squashing of speculative instructions and more frequent stalls for long-latency correction. SEA also requires the
Table 2.3: Operating point parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Nominal Baseline</th>
<th>Deep Pipe</th>
<th>SEA</th>
<th>CP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Frequency</td>
<td>Low Vdd</td>
<td>—</td>
<td>968 MHz</td>
<td>968 MHz</td>
</tr>
<tr>
<td></td>
<td>Nom Vdd</td>
<td>2.68 GHz</td>
<td>2.56 GHz</td>
<td></td>
</tr>
<tr>
<td>L1 $</td>
<td>Low Vdd</td>
<td>—</td>
<td>2.37 ns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nom Vdd</td>
<td>0.68 ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Singlebit Correction Latency</td>
<td>Hi-ECC</td>
<td>—</td>
<td>2 cycles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VS-ECC</td>
<td>—</td>
<td>2 cycles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BitFix</td>
<td>—</td>
<td>3 cycles</td>
<td></td>
</tr>
<tr>
<td>L1 Multibit Correction Latency</td>
<td>Hi-ECC</td>
<td>—</td>
<td>13 cycles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VS-ECC</td>
<td>—</td>
<td>13 cycles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bit-Fix</td>
<td>—</td>
<td>3 cycles</td>
<td></td>
</tr>
<tr>
<td>Pipeline Stages</td>
<td>Hi-ECC</td>
<td>7</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>VS-ECC</td>
<td>7</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Bit-Fix</td>
<td>7</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>L2 Latency</td>
<td>—</td>
<td>—</td>
<td>10 ns</td>
<td></td>
</tr>
</tbody>
</table>

additional pipeline stages that CP needs for forwarding speculative values. A fourth design point is SECDED+Disable where each word is protected by an ECC that can correct up to one error (this is the same SEC as used for our VS-ECC implementations). If more than one error is identified, that word is disabled, requiring an access to the next level in the cache hierarchy. Since SECDED requires more than one cycle of additional latency, implement it in an 11-stage pipeline. The final design point is CP. It has a slightly lower frequency at nominal voltage than the other design points and a slightly lower frequency than Deep Pipe and SEA at 650mV to account for the addition of two MUXes to the critical path of a cache access (see Figure 2.8).

2.6.3 Experimental Setup

We evaluate CP over benchmarks from the Spec2000 [30] and Spec2006 [31] benchmark suites executing on a two-issue in-order core. Core microarchitectural parameters were chosen to be similar to the ARM Cortex-A7 [32] and are enumerated in Table 2.4. An aggressive branch predictor was chosen to not unduly penalize Deep Pipe. Performance simulations were run using gem5 [33], fast forwarding for 1 billion instructions and executing for 1 billion instructions. Frequency for a given voltage was determined by the linear scaling in [11]. Nominal dynamic and static power overheads were determined using the simulation results and McPAT [34]. Low supply voltage...
Table 2.4: Basic core characteristics.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number Cores</td>
<td>1 in-order</td>
</tr>
<tr>
<td>Register File</td>
<td>32 Int, 32 FP</td>
</tr>
<tr>
<td>Fetch/Decode/Issue Width</td>
<td>2/2/2</td>
</tr>
<tr>
<td>BTB Size</td>
<td>4096 entries</td>
</tr>
<tr>
<td>RAS Size</td>
<td>16 entries</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Tournament</td>
</tr>
<tr>
<td>ALUs/FPUs/MDUs</td>
<td>2/1/1</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>64B</td>
</tr>
<tr>
<td>L1 I$</td>
<td>Nominal (1.2V, all designs)32KB, 4-way</td>
</tr>
<tr>
<td>Hi-ECC (650-840mV)</td>
<td>16KB, 2-way</td>
</tr>
<tr>
<td>VS-ECC (650-840mV)</td>
<td>32KB, 4-way</td>
</tr>
<tr>
<td>Bit-Fix (650-840mV)</td>
<td>24KB, 3-way</td>
</tr>
<tr>
<td>L1 D$</td>
<td>Nominal (1.2V, all designs)32KB, 4-way</td>
</tr>
<tr>
<td>Hi-ECC (650-840mV)</td>
<td>16KB, 2-way</td>
</tr>
<tr>
<td>VS-ECC (650-840mV)</td>
<td>32KB, 4-way</td>
</tr>
<tr>
<td>Bit-Fix (650-840mV)</td>
<td>24KB, 3-way</td>
</tr>
<tr>
<td>L2 Unified $</td>
<td>1MB, 8-way</td>
</tr>
<tr>
<td>Memory Configuration</td>
<td>2 GB of 1066 MHz DDR3</td>
</tr>
</tbody>
</table>

dynamic power was scaled quadratically with respect to voltage [8]. Low supply voltage static power was scaled cubically with respect to voltage [8]. The L2 and the main memory are not scaled (i.e., we model them to have an absolute latency in terms of ns).

2.7 Experimental Results

In this section, we evaluate the impact of the reduced error correction latency enabled by CP on processor-level performance, power, and energy.

2.7.1 Low Voltage (650mV) Operation

Figure 2.10 shows the performance of the different schemes at 650mV compared to the nominal baseline operating at 1.2V. For simplicity, we discuss Hi-ECC results here, although the trends for VS-ECC and Bit-Fix are similar. We see a 22% performance benefit over the deepened pipeline design in spite of slightly lower frequency (945MHz vs. 968MHz). This is due to a reduced penalty for branch and data hazards since the number of pipeline stages is smaller (9 vs. 11). Furthermore, the additional pipeline stages in CP continue to forward to dependent instructions, while the additional pipeline
stages in the deepened pipeline design lead to stalls for the load-dependent instructions. CP increases performance by 42% over the SECDED with word disable scheme, which performs the worst of all correction schemes on average. When compared to Speculate on Every Access (SEA), CP increases performance by 18%. There are two reasons for this performance improvement. First, CP reduces the mis-prediction rate and thus the penalty from squashing speculative instructions. Second, CP reduces the number of long-latency stalls for strong error correction because the fast, weak correction in CP can correct errors within the data word, which may reduce multi-bit errors to single-bit errors. Finally, CP achieves average performance within 13% of the ideal, zero-latency, zero-capacity correction. The slight difference in performance is due to the slightly lower frequency of operation at 650mV, stalls due to non-predictions and mis-predictions (see Table 2.1). When compared against the SECDED+Disable design point, CP has a performance benefit of 42% because SECDED requires additional pipeline stages for both L1 caches and also incurs long-latency misses to the next levels of the cache hierarchy.

Figure 2.11 shows the energy benefits that the error correction predic-
tion scheme can provide over the reduced frequency and pipeline deepening schemes. The observed benefit of CP compared to the deepened pipeline design is 19% and it occurs due to the 22% higher performance. CP has 16% lower energy than SEA because of the increased performance from correct prediction and decreased power consumption. All error correction techniques show a reduction of average energy by 58% or more relative to the nominal baseline. We report energy benefits for a core where logic and SRAMs are on a single power rail. If logic and SRAMs are on different rails, energy benefits will still be significant because SRAM energy would dominate at low supply voltages.

2.7.2 Nominal Operation

We recognize that for many applications low power/energy operating points are desirable, but they must not come at the price of hurting performance at nominal operating points. We evaluated the performance of each design point at the nominal voltage (1.2V). Figure 2.12 shows that CP has only a slight performance degradation of 1.2% when operating at the nominal voltage due to the addition of a MUX on the critical path of L1 cache accesses. This shows that CP is not only an attractive design point for fixed voltage systems running at low voltages (e.g., 650mV), but also for systems that support DVFS to provide high performance at nominal operating points. In contrast, pipeline deepening has an average performance degradation of over 15%. The high performance degradation is due to two factors (1) data and control hazards are more expensive because of increased number of pipeline stages, and (2) since the additional pipeline stages corresponding to the data cache accesses cannot forward values until the access is complete, dependent instructions have to stall. Finally, the reduced frequency design and SEA have high performance at nominal operation. However, these designs may have significant performance and energy overhead at low voltage operation (Figure 2.10, Figure 2.11).
2.7.3 Sensitivity

The benefits provided by CP are dependent on several parameters, including the voltage of operation, the latency impact of CP, and the fault rates of a given voltage. In this section we perform a sensitivity analysis of these parameters.

Our results in Section 2.7.1 were presented for 650mV. It is the lowest voltage at which all the ECC bits for a 32-bit data word can fit within another 32-bit word. In this section, we compare the performance of correction prediction against other design points over higher supply voltages and corresponding error rates. For Hi-ECC and VS-ECC, we select the weakest BCH code which guarantees reliability at that voltage equivalent to the reliability at nominal voltage. This means that the given supply voltage is the lowest supply voltage at which the design can be run (e.g., the BCH decoder that is required for 840mV operation would not be sufficient to operate at 700mV).

The CP error correction hardware is constant across all supply voltages.

Figure 2.13 shows the results. The results show that CP allows operation over a large voltage range at a performance comparable to the ideal scheme. Performance degradation of CP applied on top of Hi-ECC compared to the ideal is only 13%, 7%, and 7% at 650mV, 700mV, and 840mV respectively. The primary reason why CP performs comparably to the ideal is that a
large fraction of errors over a wide voltage range are predictable by the fast, weak correction. We also observe that while CP provides better performance than other design points over the entire range that was evaluated, the performance advantage depends on the operating voltage. For example, while CP has nearly identical performance to SEA at 840mV, the performance advantage increases to 18% at 650mV. This is because the number of errors is considerably higher at 650mV than 840mV. For SEA, this leads to significantly higher number of squashes and long-latency stalls. For CP, most of the cache accesses continue to be predicted by the fast, weak correction scheme at 650mV. Finally, CP continues to perform much better than reduced frequency and pipeline deepened designs even at higher supply voltages (e.g., 840mV). This is because the reduced frequency design has significantly lower frequency even at 840mV (1.111GHz vs. 1.524GHz). Similarly, the deepened pipeline not only suffers from the standard disadvantages of having a large number of pipeline stages (increased cost of branch and data hazards), but also has additional stalls due to the load-dependent instructions waiting for the error correction for the data cache access to complete.

The results in Section 2.7.1 correspond to a 2.5% latency overhead for CP as modeled by the methodology in Section 2.5.2. Figure 2.14 shows the performance of CP applied to Hi-ECC (the correction scheme CP performs worst with) assuming its latency overhead is double and quadruple our calcu-
lated value. Despite the increased overhead (5% and 10%), CP still increases performance by 16% and 12%, respectively.

Our results in Section 2.7.1 are based on the 65nm fault rate data from [10] (note that prior work [11, 8] used 130nm fault rate data). We explored the sensitivity of benefits to fault rates nearly 5x lower than [10]. Figure 2.15 shows that CP can still provide performance improvements of 14-20% for a lower fault rate vs. voltage curve. These results (including results in Figures 2.13, 2.14, and 2.15) attest to the generality of our approach.

The results reported above are for min-sized 6T SRAMs. Prior work (e.g., [25]) has demonstrated that circuit-level techniques such as doubling the size of transistors in 6T cells or using 8T cells may allow reduced voltage operation. However, CP can still yield performance benefits when used with such circuit-level techniques. To quantify the benefits we rely on the fault rate vs. voltage dataset from [25] and voltage versus frequency scaling dataset from [35].\footnote{We use fault rate vs. voltage data from [25] since the data set used for our main set of results in Section 2.7.1 did not have fault rate vs. voltage data for 6T upsized or 8T SRAM cells. We use the non-linear voltage vs. frequency scaling data from [35] since the linear voltage vs. frequency scaling assumption no longer applies for the low voltages allowed by 6Tx2 and 8T cells.} The 6Tx2 SRAM cells allow operation at 454mV with the evaluated strong error correction techniques. When implemented on top of 6Tx2 SRAM cells, CP can achieve a performance improvement of up to 13-
Figure 2.16: 6T 2x upsized sensitivity study using [25, 35].

20% as shown in Figure 2.16. Such upsizing results in a 33% area increase and a doubling of static power. By using 8T SRAM cells, the additional static power can be reduced, while still allowing a minimum voltage of 454mV. When implemented on top of 8T SRAM cells, CP provides up to a 13-20% performance improvement as shown in Figure 2.17. For completeness, Figure 2.18 shows CP results when applied to 6T cells using these datasets. At the lowest voltage where the reliability target can be met, CP shows a performance benefit of 12-19% for 6T transistors, similar to the primary results presented in Section 2.7.1.

2.8 Future Work and Discussions

In this chapter, we explore one implementation of correction prediction; many other implementations with different tradeoffs exist. For example, instead of using a BIST routine to populate the CPT as described previously in the chapter, one can use a learning mechanism to populate the CPT by dynamically identifying the weak memory cells at runtime. Also as an example, instead of allocating a static CPT entry to every group of four words as described previously in the chapter, one can also use a smaller CPT table with fewer dynamic entries that correspond to the most frequently accessed
words. Finally, other implementations of correction predictors are possible, such as weak error-correcting codes or history prediction.

While our evaluation is performed in the context of an L1 cache, the concept of correction prediction is applicable to other on-chip memories. Examples include L2 and L3 caches, as well as non-cache on-chip memories such as GPU register files and the memory systems of embedded processors which often reside on-chip. Correction prediction is also applicable to emerging technologies such as STT-RAMs, where fault rates are high [36]. In addition to in-order cores, correction prediction can be applied to other processor micro-architectures that support speculative execution, such as out-of-order processors and processors with runahead threads. The details needed to apply correction prediction to these different contexts are outside the scope of this dissertation.

Finally, correction prediction bears some resemblance to value prediction, which seeks to predict the value of a load before the load instruction completes. They differ in three main ways. First, value prediction is beneficial only for accesses to words with value locality, while correction prediction is not limited by this requirement. Second, value prediction benefits only load instructions, while correction prediction benefits all instructions by predicting the correct values of instruction words accessed from the I-cache. Third, by predicting the values of weak cells in a word, instead of predicting the com-

Figure 2.17: 8T sensitivity study using [25, 35].
Figure 2.18: 6T sensitivity study using [25, 35].

plete value of the whole word, correction prediction can require significantly lower overheads than value correction while providing the same prediction accuracy.

2.9 Summary

On-chip memories consume an increasingly large fraction of chip power. The reliability of on-chip memories determines their voltage and, therefore, the power these memories consume. Voltage scaling can be used to significantly reduce the power consumed by on-chip memories and chips as a whole. However, aggressive voltage scaling leads to high error rates in on-chip memories (e.g., caches). Strong error correction can be used to tolerate high error rates in on-chip memories. However, such strong error correction may require significant latency relative to the memory access itself. We propose correction prediction, a scheme that reduces the latency of strong error correction by using a fast mechanism to predict the result of strong error correction. We present CP, a fast, weak correction mechanism that predicts the result of strong error correction with a mis-prediction rate of <0.1%. This reduces the effective access latency of a 32KB, four-way SRAM L1 cache by 38%-52%. For a two-issue in-order core, CP provides 16%-21% energy reduction.
compared with using a strong error correction scheme alone, while incurring less than a 1.2% performance degradation at nominal voltage.
Chapter 3
Rescuing Uncorrectable Fault Patterns in On-Chip Memories through Error Pattern Transformation

In Chapter 2, we presented an architectural technique for improving the performance and energy of a processor by reducing the latency overhead of error correction. In this chapter, we propose error pattern transformation, a novel low-latency error correction technique that allows on-chip memories to be scaled to voltages lower than what has been previously possible while still maintaining the same reliability targets. Our technique relies on the observation that the number of on-chip memory errors that many error correcting codes (ECCs) can correct differs widely depending on the error patterns in the logical words they protect. We propose adaptively rearranging the logical bit to physical bit mapping per word according to the BIST-detectable fault pattern in the physical word. The adaptive logical bit to physical bit mapping transforms many uncorrectable error patterns in the logical words into correctable error patterns and, therefore, improving on-chip memory reliability. This reduces the minimum voltage at which on-chip memory can run by 70mV over the best low-latency ECC baseline, leading to a 25.7% core-wide power reduction for an ARM Cortex-A7-like core. Energy per instruction is reduced by 15.7% compared to the best baseline.

3.1 Introduction

Current and future process technologies face serious power challenges. A well-known technique that effectively reduces processor power consumption is supply voltage scaling. One major challenge for voltage scaling is that the reliability of SRAM cells decreases with the supply voltage. As the supply voltage decreases, an rapidly increasing fraction of SRAM cells become faulty due to process variations [26, 25]. This problem becomes more pronounced at smaller feature sizes (Figure 3.1) and is expected to get worse [21]. As such,
the reliability of SRAM cells, and not the reliability of logic circuits, often determines the extent to which supply voltage can be reduced [8, 37, 38]. Although using more robust SRAM cell implementations with larger area and leakage can improve the reliability of SRAMs at low supply voltages, a significant fraction of these SRAM cells are still faulty at low voltages, as shown in Figure 3.2.

Many recent works have investigated using an error correcting code, or ECC, to tolerate the high fault rates of SRAMs at low supply voltages [9, 11, 39, 13]. However, there is often a strong trade-off between error correction coverage and latency between different ECC schemes. For example, Figure 3.3 shows that the four-bit-correcting BCH (127,64) ECC can tolerate many factors higher bit failure rates than weaker ECC schemes for the same coverage; however, this comes at the cost of incurring up to 20X higher latency overhead.\footnote{BCH decoder latency is taken from [19]; the calculation assumes an oracular BCH decoder whose latency depends on the actual number of faults in a word, as opposed to having a constant worst-case latency.} Although stronger ECC schemes provide the high error correction coverage needed to enable low supply voltage, their high error correction latencies make them unattractive for on-chip memories, where latency is often critical.

In this chapter, we propose error pattern transformation, a novel low-latency microarchitectural technique that allows on-chip memories to be scaled to voltages lower than what has been previously possible. We observe that although many ECCs only guarantee correction of a few memory errors, they can opportunistically correct more errors depending on the error pattern in the logical words they protect. As such, we propose transforming the uncorrectable error patterns in logical words into correctable ones by intelligently rearranging the logical bit to physical bit mapping when storing a logical word into a physical word in an on-chip memory. This improves on-chip memory reliability and, therefore, leads to a reduction in the minimum voltage at which on-chip memory can be run reliably.

Error pattern transformation (EPT) is a general technique that can be applied on top of many prior works on improving on-chip memory reliability. EPT can provide reliability benefits even in the presence of soft errors and erratic faults. Our evaluations show that EPT reduces the minimum voltage at which on-chip memory can run by 70mV over the best low-latency
Figure 3.1: SRAM failure probability w.r.t. to voltage for different processing technologies [26, 25, 40]. Technologies with smaller feature sizes exhibit higher fault rates.

Figure 3.2: 65nm SRAM failure probability w.r.t. to voltage for different SRAM cells [25]. Larger SRAMs have lower but still significant fault rates at low voltages.
3.2 Motivation

The key insight driving the work in this chapter is that for many ECCs, the number of errors that they can correct differs widely depending on the observed error pattern (i.e., the bit locations of the individual errors). Consider, for example, a segmented ECC, which breaks a word into several independent and equally sized segments, each with its own check bits for error correction [9]. The number of errors that a segmented ECC can correct

Figure 3.4: Uncorrectable (above) and correctable (below) error patterns in a segmented ECC that corrects one error per segment. Low-latency ECCs can correct different numbers of errors for different error patterns.
depends on how many errors are located in each segment, as illustrated in Figure 3.4.

Our second example ECC does not break a word into independent segments. An Orthogonal Latin Square Code, OLSC(128,64), protects 64 bits of data with 128-64=64 bits of redundancy. OLSC(128,64) performs error correction in multiple levels of majority voting, where each level consists of multiple majority voters processing overlapping sets of input bits in parallel for low-latency error correction. When there are too many errors in the inputs to one of the majority voters, the output of the voter can flip; this can in turn flip the outputs of subsequent voting levels. As such, the number of errors that the OLSC ECC can correct again depends on the error pattern. OLSC(128,64) can correct up to 32 bad bits among the 128 bits for some specific error patterns; for all error patterns in general, however, OLSC(128,64) only guarantees correction of up to four bad bits.

In fact, it is quite common for many low-latency ECCs (the latency-coverage trade-offs of different ECCs are shown in Figure 3.3) to have some error patterns for which more errors can be corrected than other error patterns. Figure 3.5 shows the fraction of t-bit error patterns that are correctable under different ECCs. Figure 3.5 considers 64-bit data words; it presents the unsegmented OLSC(128,64) ECC as well as a segmented Hamming(7,4) ECC and a segmented OLSC(8,4) ECC, where a segmented ECC(n,k) breaks a 64-bit data word into segments of k data bits with n – k check bits per segment. Figure 3.5 shows that a significant fraction of t-bit error patterns are correctable for a large range of t even though segmented Hamming(7,4),
In this chapter, we improve the coverage of low-latency ECCs by adaptively transforming an uncorrectable BIST-detectable t-bit error pattern in a logical word into one of the many correctable t-bit error patterns (the latency-coverage trade-offs of different ECCs are shown in Figure 3.3). This allows low-latency ECCs to obtain similar coverage as high-latency ECCs (e.g., the coverage of the BCH(127,64) shown in Figure 3.5), while incurring only a small fraction of the latter’s long latency overhead. The improved coverage allows on-chip memories to be scaled to voltages lower than what has been previously possible.

3.3 Error Pattern Transformation

Consider the proposed technique in context of a cache. We observe that the same physical fault pattern in a cache word can manifest as different error patterns in the logical word presented to the ECC decoder depending on the mapping of the logical bits in a cache word to the physical bits stored in the physical cache word. For example, the logical bit to physical bit mapping from a logical word to a physical cache word, or simply *bit ordering*, shown in the left half and the right half of Figure 3.6 generates error vectors ‘01010000’ and ‘10000001’, respectively, even though the physical fault pattern in the

Figure 3.6: Possible logical to physical bit orderings. The error pattern in a logical word depends on the bit ordering used to access the physical word.

segmented OLSC(8,4), and OLSC (128,64) guarantee correction of only 1, 1, and 4 errors, respectively.
Figure 3.7: The physical fault coverages of low-latency ECCs with different numbers of bit orderings. With multiple bit ordering options, the low-latency ECCs approach, and sometimes, exceed the slow BCH ECC in coverage.

cache word remains the same. We seek to transform uncorrectable error patterns in the logical words into correctable error patterns by supporting multiple bit orderings per physical cache word. In comparison, conventional cache designs support only a single bit ordering, which maps bit x in a logical word (e.g., logical bit 0) to the same bit x in the physical cache word (e.g., physical bit 0).

When accessing a cache word with a known (BIST-detectable) physical fault pattern, our proposal uses a pre-recorded bit ordering that generates only correctable errors for the known physical fault pattern in the cache word; the pre-recorded bit ordering is selected from a pool of available bit orderings. Under our proposal, a physical fault pattern in a cache word is uncorrectable only if no bit ordering that generates only correctable error patterns for the given physical fault pattern can be found; intuitively, the larger the number of supported bit orderings, the less likely that a bit ordering that generates only correctable error patterns cannot be found for a given physical fault pattern.

Figure 3.7 shows the calculated ideal fault coverages for different ECCs
under different numbers of supported bit orderings. Figure 3.7 shows that the fraction of correctable fault patterns increases significantly as the number of supported bit orderings increases. In fact, for a sufficient number of available bit orderings (e.g., 32 to 256 bit orderings), coverage approaches, and sometimes, exceeds the high-latency BCH ECC. This improved coverage can be used to scale voltages lower than what has been previously possible for cache memories.

Our proposal requires addressing three main challenges. First, how to identify the appropriate bit ordering for each cache word. Second, how to record the identified bit orderings in a space-efficient manner. Third, how to enact the appropriate bit ordering during cache accesses at low latency overhead. We address these challenges in Section 3.3.1 through Section 3.3.3. Finally, Section 3.3.4 walks through examples of how to apply our proposal.

3.3.1 Bit Ordering Selection

Low voltage SRAM faults are largely hard faults that can be identified using a BIST (Built-in Self-Test) routine that is run either during post-manufacture testing or at set intervals during processor lifetime [8, 11]. The following describes a general approach, applicable to different ECCs, for selecting the appropriate bit ordering that only generates correctable errors for a physical fault pattern that has been identified in a cache word.

The first step is to identify the physical fault vector of a cache word at low supply voltage using BIST. Next, the fault vector is sent to the bit reordering logic (described in detail in Section 3.3.3), which modifies the ordering of the bits in the fault vector. The modified fault vector is then fed into a correctability checker to check whether all possible error patterns due to the modified fault vector are correctable. The correctability checker is specific to an ECC. For example, for the segmented Hamming (7,4) ECC,

\[ f(t) \]

Coverages are determined as follows. Let \( f(t) \) be the fraction of \( t \)-bit error patterns that are uncorrectable by an ECC scheme; we obtained \( f(t) \) for each of the fast ECC implementations via Monte Carlo simulations. Given that there are \( n \) ways to reorder the logical bits stored in a physical word, a \( t \)-bit fault pattern is uncorrectable if the fault pattern expresses uncorrectable error patterns under all \( n \) bit orderings. The probability that a \( t \)-bit fault pattern is expressed as an uncorrectable error pattern under all \( n \) bit orderings is \( f(t)^n \), assuming that the \( n \) orderings are generated independently from one another. The fraction of \( t \)-bit fault patterns that are correctable given \( n \) bit orderings is, therefore, \( 1 - f(t)^n \).
the correctability checker counts the number of ‘1’ s in every segment of seven
adjacent bits in the fault vector under test; the checker asserts a fail signal if
any segment contains more than one ‘1’ and asserts a pass signal otherwise.
If the check passes, the tested bit ordering for the cache word is recorded
so that the ordering will be used to access that cache word for all future
accesses. Otherwise, the bit reordering logic checks another bit ordering of
the identified fault vector until one of the bit orderings passes or until all
available bit orderings have been tested; in the latter case, the cache word is
reported as having an uncorrectable fault pattern.

Figure 3.8 summarizes the steps described above. Note that all of the steps
only have to be performed once for each cache word (e.g., once during post-
manufacturing testing). They are not performed during normal operations
and, therefore, do not affect runtime performance. Assuming a 50s BIST
overhead to identify all fault patterns for a 2MB cache [11], our 32KB L1
cache requires 0.78s. Assuming the worst case of $2^8$ OID calculations per word
and two cycles per calculation (one cycle to generate a new bit ordering and
one cycle to check the new ordering), a 32KB L1 cache requires only 131K
cycles. Therefore, selection of OID bits would add a negligible amount of
time to the BIST routine.

3.3.2 Tracking Bit Orderings

To support $2^k$ different bit orderings per cache word, we allocate k bits per
physical cache word to record the chosen bit ordering for each physical cache
word. We call these k bits per cache word the Ordering ID or OID of the
physical cache word. We store the OIDs in the tag array where they are
accessed at the same time as the tags. Similar to prior work [11], a copy
of the OIDs is stored off-chip to allow the OIDs to be used across multiple
switches to low voltage or after the chip is powered down.

3.3.3 Bit Reordering and Order Recovery

Prior to a write to a physical cache word, we reorder the bits in the logical
word to be written according to the recorded OID of the cache word. Con-
versely, after reading from a cache word, the earlier modification to the bit
Figure 3.8: Selecting the appropriate bit ordering for a cache word. A BIST circuit generates the fault pattern of a physical word. Logical bit reorderings are tried until one guarantees that the logical word will be correctable. In our evaluations, this takes less than two attempts on average.
ordering has to be reversed to recover the original logical word. Figure 3.9 details the steps needed for accessing a cache word. During a cache access, the OID is read at the same time as the tag; however, bit reordering and order recovery are on the critical paths of cache writes and reads, respectively. It is critical that these two steps are implemented at very low latency.

To provide low latency bit reordering and order recovery, we evenly divide each logical word into multiple small groups of bits and perform bit reordering and order recovery on all groups in parallel; bits in each parallel reordering group (or simply, PRG) can only be moved within the PRG but not across PRG. Intuitively, the smaller the PRG sizes, the higher the parallelism in the reordering and order recovery logic, and therefore, the lower the latency of the bit reordering and order recovery logic. However, the smaller the PRG sizes, the less flexible bit reordering becomes, which limits the correctable fault coverage because it reduces the number of correctable error patterns that can be exploited. As such, the size of the PRGs needs to be selected based on the underlying ECCs. Section 3.3.4 walks through some examples of PRG selection and sizing.

A straightforward way to reorder the bits in a PRG is to use a set of static circuits, each designed to reorder bits within a PRG in a specific way. However, this can lead to a large area overhead when supporting a large number of bit orderings per cache word. Instead, we use barrel shifters as...
a more scalable approach to reorder a PRG. A barrel shifter can generate a different bit ordering within a PRG for a different shift distance input value. In addition to providing good scalability, the barrel shifters can also reorder the bits in a PRG at low latency since the PRGs are small (e.g., only 8 to 16 bits per PRG). Synthesis shows (details in Section 3.4.1) only < 0.3 cycle of barrel shifting latency assuming 20 FO4s per cycle.

Figure 3.10 illustrates the details of how to implement the bit reordering and order recovery logic using barrel shifters. The barrel shifters in the bit reordering logic and order recovery logic differ simply by the direction of rotation so that the actions taken by the former can be reversed by the latter. The shift distance inputs to the barrel shifters are taken from the OID of a cache word; this allows a logical word to be reordered/recovered according to the OID of the cache word that stores the logical word. Note that the number of bits in the shift distance input to a barrel shifter is a function of the size of each PRG; this number may or may not be equal to the size of an OID. As such, each barrel shifter takes as its shift distance input a combination of bits from the OID. Different barrel shifters take different combinations of the bits from an input OID to reduce the correlation between different PRGs since this correlation can reduce the effectiveness of bit reordering by reducing the movement of the bits in a logical word relative to each other. For our evaluation in Section 3.5, we choose the combinations of input bits to the barrel shifters experimentally by picking the set of combinations that provides the best empirical fault coverage out of ten randomly generated sets of combinations.

3.3.4 Application Examples

Error pattern transformation (or EPT) is a general technique that can be applied on top of many prior works on improving on-chip memory reliability. In this section, we describe how to apply EPT to two different ECCs.

Applying to Segmented ECCs

For segmented ECCs, we construct each PRG using logical bits from different segments to move errors in segments with more errors into segments with fewer errors; this PRG composition principle benefits all segmented ECCs.
Figure 3.10: Bit reordering and order recovery logic. PRG stands for a Parallel Reordering Group.

Figure 3.11 shows the PRG composition for a 16-bit data word protected by the segmented Hamming(7,4) ECC; each column of squares represents a PRG while each row of squares represents a segment. Figure 3.11 also illustrates how the composition of each PRG from bits in different segments improves the fault coverage of a segmented ECC.

Applying to Unsegmented ECCs

For unsegmented ECCs, a good PRG composition may be different for different unsegmented ECC implementations. As such, we propose a two-stage EPT that provides more freedom in bit reordering to better suit the specific needs of different unsegmented ECCs. During the first stage, each PRG is composed from bits in the same row so that the bits can be reordered along the X-axis. During the second stage, each PRG is composed from bits in the same column so the bits can be reordered along the Y-axis. Figure 3.12 shows the PRG composition for a 16-bit data word protected by the OLSC(24,16) ECC.

Section 3.6 provides more examples of application of EPT to prior works
(a) A bit ordering that can generate uncorrectable errors for segmented Hamming(7,4).

(b) A bit ordering that only generates correctable errors for segmented Hamming(7,4).

Figure 3.11: PRGs for segmented Hamming(7,4). Each row corresponds to one segment within the access granularity. Green and blue represent data and ECC bits, respectively. Red represent bits mapped to faulty SRAM cells. All rows are accessed at once, are decoded at once, and are combined to form one word.

on improving on-chip memory reliability.

(a) A bit ordering with uncorrectable errors.

(b) Bit ordering after stage 1 reordering.

(c) Bit ordering after 2D reordering.

Figure 3.12: PRGs for OLSC(24,16). Green and blue represent data and ECC bits, respectively. Red grid represents bits mapped to faulty SRAM cells.
3.4 Methodology

3.4.1 EPT-Based Cache Resilience Designs

To demonstrate the applicability of EPT to a variety of ECCs that can be used to protect deeply voltage scaled on-chip memories, we apply EPT to three different ECCs in the context of a 32KB L1 cache with a word size of 64 bits - the segmented Hamming(7,4), segmented OLSC(8,4), and OLSC(128,64). All three ECCs protect at the granularity of a 64-bit data word, which is the word size of the evaluated L1 cache. For sensitivity analysis, we evaluate EPT using both 5-bit OIDs and 8-bit OIDs; we refer to them as 5-bit EPT and 8-bit EPT, respectively. We apply EPT to segmented Hamming(7,4) and segmented OLSC(8,4) by adding one level of 16-bit barrel shifters and apply EPT to OLSC(128,64) by adding two levels of 16-bit barrel shifters.

To accurately characterize the latency and area overheads of EPT, we implemented these ECCs, both by themselves and with EPT, in RTL and synthesize them using Synopsys Design Compiler [41] and the TSMC 65GP standard cell library. Cadence SoC Encounter [42] was used for physical layout. Area and latency values were verified using Encounter. Table 3.1 shows the latency in cycles and area overheads normalized to the evaluated 32KB cache (see Section 3.4.3); our evaluation assumes the 20 FO4 cycle commonly used in prior works [8, 25]. Table 3.1 shows that all three ECCs stay within one cycle of latency, with or without EPT. By itself, each level of barrel shifters required by EPT only incurs < 0.3 cycle latency and 3%

Table 3.1: Synthesized decoder latency and area. Area overheads is w.r.t. a 0.76mm² L1 cache.

<table>
<thead>
<tr>
<th>ECC Type</th>
<th>Cycle</th>
<th>Area</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segmented Hamming(7,4)</td>
<td>0.22</td>
<td>0.44%</td>
<td>528</td>
</tr>
<tr>
<td>EPT:segmented Hamming(7,4)</td>
<td>0.43</td>
<td>1.53%</td>
<td>2009</td>
</tr>
<tr>
<td>Segmented OLSC(8,4)</td>
<td>0.16</td>
<td>0.40%</td>
<td>573</td>
</tr>
<tr>
<td>EPT:Segmented OLSC(8,4)</td>
<td>0.46</td>
<td>1.60%</td>
<td>2095</td>
</tr>
<tr>
<td>OLSC(128,64)</td>
<td>0.68</td>
<td>8.89%</td>
<td>6394</td>
</tr>
<tr>
<td>EPT:OLSC(128,64)</td>
<td>0.98</td>
<td>10.10%</td>
<td>9187</td>
</tr>
</tbody>
</table>
area overhead. In particular, the two segmented ECCs both stay within 0.5 cycle of latency, with or without EPT. The low-latency overheads justify our design choices in Section 3.3. In our evaluations, we model the latency overhead of all of the ECCs in Table 3.1 as one cycle; the one exception is EPT:OLSC(128,64), which we conservatively model as two cycles of latency. Table 3.2 lists the EPT latency overheads that are used during our performance and EPT area overheads used during our power evaluations.

### Table 3.2: Evaluated EPT overheads.

<table>
<thead>
<tr>
<th>Baselines</th>
<th>Latency Overhead (Cycles)</th>
<th>Usable Cache Size</th>
<th>Decoder Area Overhead</th>
<th>Metadata Overhead</th>
<th>Total Area Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-bit EPT+Segmented Hamming(7,4)</td>
<td>1</td>
<td>16KB</td>
<td>3.06%</td>
<td>0%</td>
<td>3.1%</td>
</tr>
<tr>
<td>5-bit EPT+Segmented OLSC(8,4)</td>
<td>1</td>
<td>16KB</td>
<td>3.2%</td>
<td>0%</td>
<td>3.2%</td>
</tr>
<tr>
<td>5-bit EPT+OLSC(128,64)</td>
<td>2</td>
<td>16KB</td>
<td>20.20%</td>
<td>0%</td>
<td>24.2%</td>
</tr>
<tr>
<td>8-bit EPT+Segmented Hamming(7,4)</td>
<td>1</td>
<td>16KB</td>
<td>3.06%</td>
<td>4%</td>
<td>7.1%</td>
</tr>
</tbody>
</table>

### Table 3.3: Evaluated baseline overheads.

<table>
<thead>
<tr>
<th>Baselines</th>
<th>Latency Overhead (Cycles)</th>
<th>Usable Cache Size</th>
<th>Decoder Area Overhead</th>
<th>Metadata Overhead</th>
<th>Total Area Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracular BCH(127,64) (Hi-ECC)</td>
<td>2 - 21 (3.2 avg)</td>
<td>100%</td>
<td>70%</td>
<td>0%</td>
<td>70%</td>
</tr>
<tr>
<td>VS-ECC</td>
<td>1-7 (1.9 avg)</td>
<td>16KB</td>
<td>37%</td>
<td>4%</td>
<td>42%</td>
</tr>
<tr>
<td>MS-ECC</td>
<td>1</td>
<td>16KB</td>
<td>9.4%</td>
<td>0%</td>
<td>9.4%</td>
</tr>
<tr>
<td>SECDED</td>
<td>1</td>
<td>32KB</td>
<td>10.9%</td>
<td>0%</td>
<td>10.9%</td>
</tr>
<tr>
<td>8T SRAMs</td>
<td>0</td>
<td>32KB</td>
<td>0%</td>
<td>0%</td>
<td>33.3%</td>
</tr>
<tr>
<td>WD-Original</td>
<td>1</td>
<td>16KB</td>
<td>&lt;1%</td>
<td>43.2%</td>
<td>44.2%</td>
</tr>
<tr>
<td>WD-IsoArea</td>
<td>1</td>
<td>16KB</td>
<td>&lt;1%</td>
<td>4%</td>
<td>5%</td>
</tr>
</tbody>
</table>

3.4.2 Baseline Cache Resilience Designs

To show that the proposed approach can lead to lower on-chip memory voltages than what has been previously possible, we compare against seven baselines. We compare against the SECDED (72,64) ECC commonly used in existing processors [32]. We compare against the primary implementation of MS-ECC [9], which uses the OLSC(128,64) ECC. We also compare against the circuit-level technique of using robustified 8T SRAM cells, instead of the usual 6T SRAM cells.

We compare against an Oracular BCH(127,64) ECC that guarantees correction of up to 10 bits of error per word. We consider this baseline oracular because we optimistically assume a BCH decoder that incurs the minimum
latency for the exact number of errors currently present in a logical word. The oracular decoder has the low-latency benefit of weaker BCH decoders that correct fewer errors per word while having the high error coverage benefit of always guaranteeing correction of 10 bits in a word. As such, this oracular BCH baseline represents the best coverage and latency obtainable by prior works that propose protecting caches with the BCH ECC, such as Hi-ECC [16].

We compare against word disable (WD), the best resilience scheme presented in [8] for L1 caches. WD is previously evaluated in the context of caches with 512-bit access granularity; in this context, WD combines two 512-bit cache lines into an error-free logical line by dividing the two lines into 32 chunks and remapping data to fault-free chunks among the 32 available chunks. When evaluating WD in the context of our L1 cache with a 64-bit word size, we note that dividing two 64-bit words into 32 chunks can result in a high storage overhead for recording the 32 chunks. As such, we evaluate two versions of WD, WD-Original and WD-IsoArea; the former divides two 64-bit words into 32 chunks, while the latter divides the two 64-bit words into fewer chunks such that the implementation requires the same number of extra storage bits as an 8-bit OID EPT scheme.

We compare against a VS-ECC [11] implementation where each 64-bit data word is divided into four 16-bit segments. The 64-bits of ECC are shared as 6-bit ECC chunks between the four segments. Each segment can have up to four 6-bit ECC chunks allocated to it; each additional 6-bit chunk guarantees correction of up to an additional error. For this VS-ECC implementation, we assume that the four segments each have a dedicated decoder that corrects up to a single error but shares two 4-bit correcting decoders; the single-bit correcting decoders are sufficient for segments with a single error while the multi-bit decoders are needed in case multiple errors are present in a segment. We use two instead of four 4-bit correcting decoders to optimize the decoder area for VS-ECC since the segments more commonly experience single-bit errors than multi-bit errors in our evaluation.

We calculate the latency of SECDED according to [19], which conservatively estimates decoder latency in FO4s by ignoring the wire delay. We calculate the decoder area overhead of SECDED(72,64) by counting the number of gates required according to [19] and assuming that each gate is equal in area as two SRAM cells; the latter assumption is consistent with prior works.
When calculating the latency of Oracular BCH(127,64), we use the formula in [19] to calculate the latency required to correct \(t\)-bit errors and weight that calculate latency by the probability of encountering \(t\) bits of errors in a word. For the decoder area overhead of Oracular BCH(127,64), we assume it is the same as the area overhead of a constant latency BCH(127,64), which we calculate according to [19]. We extrapolate the latencies and area overheads of MS-ECC and WD from their respective papers. We follow the methodology for the parallel BCH decoder in VS-ECC to calculate the latency and decoder area overheads of our VS-ECC adaptation; we assume that the latency overhead of VS-ECC when a cache word is accessed is determined by the segment with the most errors among the four segments in the cache word, since this segment takes the longest to correct.

3.4.3 Experimental Setup

Keeping with the context of this work - energy efficient computing, we evaluate a two-issue in-order core similar to the ARM Cortex-A7 [32]; the detailed micro-architectural parameters are enumerated in Table 3.4. The baseline core operating at nominal voltage (1.2V) uses no ECC for L1 accesses.\(^3\) For each design that requires correction, we deepen the pipeline to accommodate the minimum correction latency of that design. The core uses a common voltage rail for both its L1 caches and the rest of the core; as such, the L1 caches determine the minimum operating voltage of the entire core, in accordance with many prior works [37, 8, 9, 43]. We only voltage scale the core, but not the rest of the system, such as the memory system and lower-level caches, which we assume are on different voltage rails. To model voltage scaling, we scale frequency and leakage power with respect to voltage using the detailed 70nm scaling given in [44]; we use the common quadratic scaling to model dynamic power with respect to voltage. We model the SRAM failure probabilities using data reported in [25]. During low voltage operations, we account for the added cache access latency due to error correction by stalling each load instruction by the number of cycles needed to perform error correction. We simulate seventeen benchmarks from the SPEC2000 [30]

\(^3\)Without any loss of generality, we can easily assume that some other default ECC is used at the nominal voltage
and SPEC2006 [31] benchmark suites\textsuperscript{4} using gem5 [33]; we fast-forward each benchmark for 1 billion instructions and execute in detailed timing mode for 1 billion instructions. We model processor power with McPAT [34] using simulation outputs from Gem5. We use CACTI [22] to model cache latency and power.

We choose a simple cache organization that is commonly used by prior works [8, 9, 43] to evaluate the relative merits of EPT and the various baselines. This simple cache organization stores the check bits of a data cache word (i.e., a cache word assigned to storing data during low voltage operation) in the same cache word in a different way in the same cache set, as these two cache words are typically always accessed in parallel in L1 caches both with or without support for voltage scaling. Unlike an alternative cache organization that uses one ECC way to protect multiple data ways, this simple cache organization that allocates a dedicated ECC way for each data way does not require expensive read-modify-write operations when updating the ECC cache word. We do not investigate new approaches to store ECC bits because it is orthogonal to the core idea of EPT. All the evaluated cache resilience designs have almost equal number of check bits as data bits in each word, and thus all effectively utilize the available space in the dedicated ECC way for each data way. The only exception is the SECDED(72,64) ECC, which only provides $72 - 64 = 8$ bits of check bits per 64-bit data word. Instead of reserving half of the ways as ECC ways, we leave all four ways as data ways and expand the size of each cache word statically to store the check bits for SECDED(72,64).

Also similar to prior works [8, 43], the cache organization we evaluate uses the larger but more reliable 10T cells [25], as the tag array is only a small fraction of the total cache area.\textsuperscript{5} Similar to these prior works, we also use the more reliable tag array to store any metadata needed for the evaluated cache resilience schemes, such as the OID bits for EPT, the disable bits for WD, and the bits to identify how many ECC chunks are allocated to each segment for VS-ECC. Note that since half of the ways in the cache are used to store ECC bits during low voltage operation, half of the tags in the tag array are unused during low voltage operation. Therefore, we reuse the unused tag

\textsuperscript{4}We evaluated ammp, applu, apsi, art470, facerec, gromacs, lbm, libquantum, lucas, mcf2006, mesa, mgrid, milc, omnetpp, soplex, swim, and wupwise.

\textsuperscript{5}Our reliability calculations also account for the failure probability of the 10T cells.
Table 3.4: Microarchitectural parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Type</td>
<td>In-order, 7-Stage</td>
</tr>
<tr>
<td>Register File</td>
<td>32 Int, 32 FP</td>
</tr>
<tr>
<td>Fetch/Decode/Issue Width</td>
<td>2/2/2</td>
</tr>
<tr>
<td>BTB Size</td>
<td>4096 entries</td>
</tr>
<tr>
<td>RAS Size</td>
<td>16 entries</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Tournament</td>
</tr>
<tr>
<td>ALUs/FPUs/MDUs</td>
<td>2/1/1</td>
</tr>
<tr>
<td>Cache Line Size</td>
<td>64B</td>
</tr>
<tr>
<td>L1 I$ Nominal (1.2V)</td>
<td>32KB, 4-way, 2-cycle</td>
</tr>
<tr>
<td>L1 I$ $\text{V}_{\text{min}}$</td>
<td>16KB, 2-way, 2-cycle</td>
</tr>
<tr>
<td>L1 D$ Nominal (1.2V)</td>
<td>32KB, 4-way, 2-cycle</td>
</tr>
<tr>
<td>L1 D$ $\text{V}_{\text{min}}$</td>
<td>16KB, 2-way, 2-cycle</td>
</tr>
<tr>
<td>L2 Unified $</td>
<td>1MB, 8-way, 20ns latency</td>
</tr>
<tr>
<td>Memory Configuration</td>
<td>1066 MHz DDR3</td>
</tr>
</tbody>
</table>

bits to store the metadata during low voltage operation. When the metadata bits do not completely fit in the unused tags (i.e., more than 5-bit OIDS), we expand the size of the tag entries in the ECC ways to accommodate the remaining metadata bits. The area overhead due to expanding the size of the tag entries are given in Table 3.3 and Table 3.2. We do not study OIDs smaller than five bits because they do not reduce area and latency overheads, but reduce coverage.

3.5 Results

In this section, we demonstrate that the proposed technique allows memory to run at lower voltages than was previously possible for a given yield target. We also discuss the core-wide power and energy benefits this entails compared to the different baselines.

3.5.1 Yield

We define cache yield as the fraction of caches where all of the data cache words in the data ways (i.e., the two data ways in our evaluated four-way set-associative caches) are functional and error-free, similar to prior works [8, 9].

\[^6\]Cache word error rates are calculated analytically where possible (e.g., Oracular BCH(127,64)). For the rest of the designs, in particular, EPT, word error rates are determined through Monte Carlo simulations.
Figure 3.13: Yield for 32KB four-way cache implemented in 6T SRAM cells. Cache yield vs. voltage for the various cache resilience schemes. The red line depicts the 99.9% yield.

Figure 3.13 shows the cache yield at low supply voltage for the different cache resilience designs. Figure 3.13 shows that for a 99.9% yield target, which is commonly used in prior works [8, 9], 8-bit EPT + Segmented Hamming(7,4) reduces the latter’s $V_{\text{min}}$ by 134mV. For the same yield target, 8-bit EPT + Segmented Hamming(7,4) reduces the $V_{\text{min}}$ of the two weakest baselines in terms of $V_{\text{min}}$ (i.e., SECDED(72,64) and MS-ECC) by 70mV or more.

After 8-bit EPT + Segmented Hamming(7,4), the next closest resilience design in terms of achieving the lowest $V_{\text{min}}$ is the 10-bit correcting Oracular BCH(127,64) ECC (the strongest version of a capacity-efficient code whose redundant bits fit in the disabled way). While 8-bit EPT + Segmented Hamming(7,4) achieves only 1mV lower $V_{\text{min}}$ than Oracular BCH(127,64), the former’s main advantage over the latter is having lower latency overhead; the 8-bit EPT + Segmented Hamming(7,4) incurs a constant one cycle latency (see Table 3.1), while the Oracular BCH(127,64) incurs 2 to 21 cycles of latency overhead per cache access (3.2 cycles, on average). A second benefit of 8-bit EPT + Segmented Hamming(7,4) over Oracular BCH(127,64) is having much lower decoder area. The area overhead of a BCH decoder increases rapidly with the word size and the number of errors to correct [19]. The 8-bit EPT + Segmented Hamming(7,4) requires both minimal word size (i.e., only seven bits per segment) and minimal correction strength (i.e., only corrects one error per segment), as compared to BCH (127,64) which uses a word size that is roughly 16X as large and corrects 10X as many errors per word.

Following after Oracular BCH(127,64), the next closest baseline in $V_{\text{min}}$
is VS-ECC. The 8-bit EPT + Segmented Hamming(7,4) only reduces \( V_{\text{min}} \) by 13mV compared to VS-ECC. However, the main benefit of 8-bit EPT + Segmented Hamming(7,4) over VS-ECC is against for having lower latency overhead; VS-ECC incurs one to seven cycles of latency overhead per cache access (1.9 cycles, on average). The 8-bit EPT + Segmented Hamming(7,4) also incurs much lower decoder area overheads, since the latter uses a word size 8X as large and corrects 4X as many errors per word.

After VS-ECC, the next closest baseline in \( V_{\text{min}} \) is WD–Original. The 8-bit EPT + Segmented Hamming(7,4) reduces the \( V_{\text{min}} \) of WD–Original by 33mV. However, WD–Original requires 4X as many metadata bits as 8-bit EPT, which incurs a 43% overhead in total cache area. On the other hand, the alternative WD–IsoArea implementation we evaluate, which requires the same number of metadata bits as 8-bit EPT, only provides a \( V_{\text{min}} \) of 769mV, which is nearly 100mV higher than the best EPT solution.

The next closest baseline in \( V_{\text{min}} \) are caches built using 8T SRAMs. The 8-bit EPT + Segmented Hamming(7,4) reduces \( V_{\text{min}} \) by 38mV and 59mV, respectively. However, 8T incurs high area (and thus power) overheads of 33%. Figure 3.13 shows that, although 8T SRAM cells provide a reduction in \( V_{\text{min}} \), EPT solutions provide lower \( V_{\text{min}} \) (e.g., EPT(8-bit OID) applied to segmented Hamming(7,4) provides a reduction of 59mV over 8T). However, 8T incurs high area overheads (i.e., 33% [25], respectively).

For sensitivity analysis, we also evaluated three 5-bit EPT design points. Among these three designs, 5-bit EPT + Segmented Hamming(7,4) provides the highest yield; it provides higher yield than 5-bit EPT + Segmented OLSC(8,4) because the former requires fewer check bits than the latter while correcting the same number of errors per segment as the latter. Having fewer bits per word reduces the average number of faulty bits per word and, thereby, improving yield. On the other hand, 5-bit EPT with Segmented OLSC(8,4) provides higher yield than 5-bit EPT + OLSC(128,64); this is because the fraction of \( t \)-bit error patterns that are correctable by OLSC(128,64) decreases more rapidly with respect to \( t \) than the fraction of \( t \)-bit error patterns that are correctable by Segmented OLSC(8,4), as shown in Figure 3.5. When the fraction of \( t \)-bit error patterns that are correctable is small, it becomes less likely to find a bit ordering that always generates one of the correctable \( t \)-bit error patterns in a logical word given a \( t \)-bit fault pattern in a cache word.
All of the above results hold irrespective of the core microarchitecture (e.g., in-order vs. out-of-order, issue-width, etc.) and workloads. The next section describes how the improved yield due to EPT translations into improved power and performance characteristics for the chosen core microarchitecture and workloads.

3.5.2 Power and Performance

Table 3.5 summarizes the core-wide power and performance of the EPT designs and the baselines. While 8-bit EPT + Segmented Hamming(7,4) operates at the same $V_{\text{min}}$ and, therefore, similar power as Oracular BCH(127,64), the former provides 21% higher IPC than the latter. This performance improvement is due to having much lower latency overhead than Oracular BCH(127,64) ECC. This shows that EPT is better than imply using a stronger ECC due to decoder latency and area overheads. Compared to 8-bit EPT + Segmented Hamming(7,4), VS-ECC only increases $V_{\text{min}}$ by 13mV. However, the higher decoder latency and area overhead of VS-ECC results in a high EPI overhead. The 8-bit EPT + Segmented Hamming(7,4) reduces EPI by 40.2% compared to VS-ECC.

WD-Original and WD-IsoArea provide similar IPC as EPT since WD also only incurs a constant one correction cycle latency. However, the high area overhead of WD-Original (i.e., 44.2%) results in a high power overhead; the 8-bit EPT + Segmented Hamming(7,4) reduces power by 20.5% compared to WD-Original. Although WD-IsoArea has similar area overhead as 8-bit EPT + Segmented Hamming(7,4), it has a much higher $V_{\text{min}}$; the 8-bit EPT + Segmented Hamming(7,4) reduces power by 31.1% over WD-IsoArea. MS-ECC also incurs similar latency and area overheads as EPT. However, the $V_{\text{min}}$ of MS-ECC is 70mV higher than that of 8-bit EPT + Segmented Hamming(72,64). As a result, 8-bit EPT + Segmented Hamming(72,64) reduces core-wide power by 25.7% compared to MS-ECC.

Using robustified SRAM cells can provide significant $V_{\text{min}}$ reduction without incurring any error correction latency overhead since they do not require error correction; however, robustified SRAM cells incur significant area. Due to the high area overhead as well as the higher $V_{\text{min}}$ of 8T cells, 8-bit EPT + Segmented Hamming(7,4) reduces overall core power by 28.1% compared to
Table 3.5: Performance, power, and energy normalized to nominal execution of the baseline design.

<table>
<thead>
<tr>
<th>Design</th>
<th>$V_{\text{min}}$ (mV)</th>
<th>Freq (GHz)</th>
<th>IPC</th>
<th>Power (%)</th>
<th>EPI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracular BCH(127,64)</td>
<td>671</td>
<td>0.931</td>
<td>1.035</td>
<td>24.3</td>
<td>50.5</td>
</tr>
<tr>
<td>VS-ECC</td>
<td>683</td>
<td>0.960</td>
<td>0.854</td>
<td>22.4</td>
<td>54.8</td>
</tr>
<tr>
<td>8T SRAMs</td>
<td>729</td>
<td>1.061</td>
<td>1.236</td>
<td>26.6</td>
<td>40.5</td>
</tr>
<tr>
<td>WD–IsoArea</td>
<td>769</td>
<td>1.141</td>
<td>1.171</td>
<td>27.7</td>
<td>41.5</td>
</tr>
<tr>
<td>WD–Original</td>
<td>703</td>
<td>1.006</td>
<td>1.228</td>
<td>24.0</td>
<td>38.9</td>
</tr>
<tr>
<td>SECDED</td>
<td>823</td>
<td>1.256</td>
<td>0.879</td>
<td>34.5</td>
<td>62.6</td>
</tr>
<tr>
<td>Segmented Hamming(7,4)</td>
<td>804</td>
<td>1.210</td>
<td>1.147</td>
<td>31.5</td>
<td>45.4</td>
</tr>
<tr>
<td>MS-ECC</td>
<td>740</td>
<td>1.083</td>
<td>1.191</td>
<td>25.7</td>
<td>39.8</td>
</tr>
</tbody>
</table>

EPT-Based Designs

<table>
<thead>
<tr>
<th>Design</th>
<th>$V_{\text{min}}$ (mV)</th>
<th>Freq (GHz)</th>
<th>IPC</th>
<th>Power (%)</th>
<th>EPI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-bit EPT+ OLSC(128,64)</td>
<td>697</td>
<td>0.993</td>
<td>0.918</td>
<td>22.0</td>
<td>48.3</td>
</tr>
<tr>
<td>5-bit EPT+ Seg OLSC(8,4)</td>
<td>693</td>
<td>0.984</td>
<td>1.229</td>
<td>20.7</td>
<td>34.3</td>
</tr>
<tr>
<td>5-bit EPT+ Seg Ham(7,4)</td>
<td>680</td>
<td>0.953</td>
<td>1.242</td>
<td>19.7</td>
<td>33.2</td>
</tr>
<tr>
<td>8-bit EPT+ Seg Ham(7,4)+</td>
<td>670</td>
<td>0.929</td>
<td>1.252</td>
<td>19.1</td>
<td>32.8</td>
</tr>
</tbody>
</table>

using 8T SRAM cells. Due to this power overhead, 8-bit EPT + Segmented Hamming(7,4) has a 19.1% lower EPI than 8T cells. Note that unlike architectural techniques where most of the circuits (e.g., decoders) needed for low voltage operations can be power gated during nominal voltage operations, caches with robustified SRAM cells continue to incur high overheads at nominal operation. For example, caches with the 8T SRAM cells incur a 12.0% core-wide EPI overhead at nominal voltage operation, whereas 8-bit EPT + Segmented Hamming(7,4) incurs only a 7.8% EPI overhead at nominal voltage (due to the overheads of the added correction pipeline stages).

The above results show that EPT is very effective at reducing the $V_{\text{min}}$ and, therefore, power of low latency caches by increasing the coverage of an ECC-based error resilience scheme. The results also show that EPT does not have some of the limitations that hamper the efficacy of other comparable techniques (e.g., high power and area overheads during nominal and low voltage operation). This makes EPT a promising technique to allow deeply voltage scaled processors and on-chip memories.
3.5.3 Additional Sensitivity Analysis

BIST-Undetectable Faults

EPT targets faulty SRAM cells that can be identified during BIST. However, some SRAM faults, such as soft errors, erratic faults, or aging related faults, cannot be detected by BIST. EPT does not protect against these faults because they cause errors to occur randomly in different locations. For L1 caches, existing processors typically only detect these errors using parity or protect against these errors using SECDED to guarantee correction of a single random error per word [32]. One can also guarantee correction of a single random error per word in caches with EPT by adding an independent layer of random error correcting code.

As an example, when storing a 64-bit dataword, besides protecting it with EPT + Segmented Hamming(7,4), one can also protect it using an independently calculated single-symbol-correcting (SSC) Reed-Solomon (RS) ECC with 5-bit symbols; each adjacent four data bits in the 64-bit dataword is mapped to a symbol in the SSC ECC when calculating the SSC ECC. When a random error occurs in a cache word, EPT + Segmented Hamming(7,4) may not be able to correct the segment in which the error resides. However, since only a single segment and, therefore, a single symbol contains the random error, the erroneous segment is guaranteed to be correctable via the SSC ECC if the random error is the only random error in the cache word.

The overhead required by this additional layer random error correction is small. The RS ECC requires 10 additional ECC bits, which fit within the $64 - (7 - 4) \cdot 16 = 16$ unused bits per ECC way of Segmented Hamming(7,4). The area overhead of the RS ECC encoder [45] and a ROM-based decoder, which stores the correctable syndromes and the error pattern corresponding to each correctable syndrome, is only 1.8% of the L1 cache area. Our evaluation shows that including this additional layer of SSC ECC to 8-bit EPT + Segmented Hamming(7,4) only increases the latter’s core-wide power by 6.8%. Since random errors only need to incur any correction latency when an random error occurs [32], which is rare, the energy overhead of adding the SSC ECC is minimal (i.e., 4.4%). In comparison to VS-ECC, the best prior

---

For simplicity, we protect against known bit faults among these 10 bits per cache word by only using EPT to move the faulty bits away from them but not storing Hamming(7,4) ECC bits for them.
Table 3.6: L2 6T performance, power, and energy normalized to nominal execution.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Vmin (mV)</th>
<th>Freq (GHz)</th>
<th>IPC</th>
<th>Power (%)</th>
<th>EPI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracular BCH(767,512)</td>
<td>701</td>
<td>1.002</td>
<td>0.956</td>
<td>28.0</td>
<td>58.5</td>
</tr>
<tr>
<td>MS-ECC</td>
<td>758</td>
<td>1.119</td>
<td>0.996</td>
<td>29.0</td>
<td>52.0</td>
</tr>
<tr>
<td>8-bit EPT+ Seg OLSC(128,64)</td>
<td>689</td>
<td>0.974</td>
<td>0.983</td>
<td>24.0</td>
<td>50.2</td>
</tr>
<tr>
<td>VS-ECC</td>
<td>754</td>
<td>1.111</td>
<td>0.967</td>
<td>30.8</td>
<td>57.3</td>
</tr>
<tr>
<td>8-bit EPT: Seg BCH(80,64)</td>
<td>742</td>
<td>1.087</td>
<td>0.930</td>
<td>29.0</td>
<td>57.4</td>
</tr>
</tbody>
</table>

Note that in our evaluation in Section 3.5.1 and Section 3.5.2, we dedicate all the error correction resources of all evaluated resilience schemes to protecting against BIST-detectable faults; in other words, additional overheads will be required for all the evaluated baselines to guarantee correction of a single random error per word while maintaining the same strength of protection against BIST-detectable faults that the baselines enjoy in Section 3.5.1 and Section 3.5.2. As such, although we do not evaluate adding random error protection for the baselines, we expect the merits of EPT relative to the various baselines to remain roughly the same as those in Section 3.5.1 and Section 3.5.2 even in scenarios where protection against random errors are needed.

L2 Caches

While EPT is primarily a technique to allow deeply voltage scaled on-chip memories, it can also be viewed as a technique to reduce the latency of strong error correction since it allows low-latency ECCs to attain the coverage of high-latency (Section 4.2 and Section 3.3). While latency is less of a concern for L2 caches, we demonstrate that EPT can still provide some benefits. Table 3.6 shows the performance, power, and energy evaluations for several designs where the L2 is co-scaled with L1s and the core.
Table 3.7: 28nm/32nm performance, power, and energy at $V_{\text{min}}$ normalized to those at nominal voltage (1.2V).

<table>
<thead>
<tr>
<th></th>
<th>$V_{\text{min}}$ (mV)</th>
<th>Freq (GHz)</th>
<th>IPC</th>
<th>Power (%)</th>
<th>EPI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracular BCH(127,64)</td>
<td>674</td>
<td>0.326</td>
<td>0.977</td>
<td>15.5</td>
<td>44.6</td>
</tr>
<tr>
<td>VS-ECC</td>
<td>688</td>
<td>0.344</td>
<td>0.896</td>
<td>14.7</td>
<td>43.5</td>
</tr>
<tr>
<td>WD–IsoArea</td>
<td>738</td>
<td>0.415</td>
<td>1.157</td>
<td>17.3</td>
<td>32.9</td>
</tr>
<tr>
<td>WD–Original</td>
<td>701</td>
<td>0.361</td>
<td>1.185</td>
<td>15.5</td>
<td>33.0</td>
</tr>
<tr>
<td>SECDED</td>
<td>777</td>
<td>0.469</td>
<td>0.851</td>
<td>21.3</td>
<td>48.8</td>
</tr>
<tr>
<td>MS-ECC</td>
<td>719</td>
<td>0.388</td>
<td>1.165</td>
<td>15.9</td>
<td>32.3</td>
</tr>
<tr>
<td>EPT-Based Designs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-bit EPT+ OLSC(128,64)</td>
<td>696</td>
<td>0.355</td>
<td>0.873</td>
<td>14.2</td>
<td>41.8</td>
</tr>
<tr>
<td>5-bit EPT+ Seg OLSC(8,4)</td>
<td>693</td>
<td>0.351</td>
<td>1.186</td>
<td>13.3</td>
<td>29.3</td>
</tr>
<tr>
<td>5-bit EPT+ Seg Ham(7,4)</td>
<td>682</td>
<td>0.336</td>
<td>1.186</td>
<td>12.6</td>
<td>28.9</td>
</tr>
<tr>
<td>8-bit EPT+ Seg Ham(7,4)</td>
<td>673</td>
<td>0.324</td>
<td>1.202</td>
<td>12.1</td>
<td>28.5</td>
</tr>
</tbody>
</table>

these designs, the L2 determines the $V_{\text{min}}$. The values in the table include the power consumed by the L2. BCH, MS-ECC, and EPT + Segmented OLSC(128,64), continue to use an ECC way for each data way, which is consistent with our evaluation of the L1 cache. Compared against MS-ECC, 8-bit EPT + Segmented OLSC(128,64) reduces combined core and L2 power by 17%.

Since the capacity overhead requirement may be more stringent at the L2 level, we also demonstrate the benefits of EPT while disabling only one of 8 ways. For this capacity overhead requirement, we compare EPT against a VS-ECC solution which has the same number of redundant bits as 8-bit EPT. This VS-ECC solution breaks each cache line into four words each of which can correct up to four faults. Across these four words, up to eight total faults may be corrected, requiring 72 bits of redundancy. We do not allocate any correction resources to BIST-undetectable faults. Results show that the energy characteristics of EPT and VS-ECC are comparable. However, EPT may still be preferable since VS-ECC decoders have high area overhead—10% of the L2 cache area. High decoder area overhead comes from implementing four 4-bit correcting BCH decoders in a fully combinational manner (i.e., lowest latency BCH decoder). The 8-bit EPT + Segmented BCH(80,64), on the other hand, only incurs a 3.5% total area overhead.
Figure 3.14: Applying EPT to Archipelago [12]. Left: Two chunks need to be remapped while the sacrificial line only has room for one chunk. Right: Only one chunk needs to be remapped after bit reordering.

28nm/32nm Technology Node

The performance, power, and energy benefits provided by fault tolerant mechanisms for low-$V_{\text{min}}$ caches are strongly tied to the characteristics of a particular technology process (i.e., how fault rates, frequency, and power scale with voltage). In this section, we explore the benefits for a 28nm fault rate [40] and 32nm frequency and power rates [38]. The 8-bit EPT + Segmented Hamming(7,4) has 17.7% lower power and 34.5% lower energy than VS-ECC (the next lowest power correction technique). The 8-bit EPT + Segmented Hamming(7,4) has 23.9% lower power and 11.8% lower energy than MS-ECC (the next lowest energy correction technique). These 28nm/32nm results demonstrate that EPT can provide voltage scaling benefits across technology nodes.

3.6 Related Work

This section describes several additional related works. Parachute [39] explores how to protect low-level caches operating at low voltages using Turbo codes. By themselves, Turbo codes under-utilize cache capacity because they

\footnote{We could not find any 28nm or 32nm 8T SRAM fault rates.}
do not increment in size in powers of two. To better utilize cache capacity, Parichute fills up the ECC ways by constructing additional check bits to the original Turbo codes and, thereby, improves the strength of the original Turbo codes as well. However, the latency overheads of Turbo codes are still significant (e.g., > 4 cycles, on average [39]). In addition, Turbo codes can correct a different number of errors depending on the error pattern in the word. As such, EPT can be applied on top of Parichute and is, therefore, orthogonal to Parichute.

Archipelago [12] avoids accessing faulty bits by remapping groups of logical bits (called chunks) from one physical cache line to a second cache line, called a sacrificial line. However, remapping chunks across lines require significant design complexity. For each remapped chunk, Archipelago records the original location of the chunk, its corresponding sacrificial line, and the corresponding position within the sacrificial line; this requires adding two additional address translation tables to the critical path of cache access [12]. In addition, a complete cache access now has to wait for two cache line accesses to complete. Since the sacrificial cache line is often not in the same cache set as the data line, to allow both cache line accesses to complete around the same time, the number of banks in the cache has to be doubled from what is needed to meet the fetch and issue width [12]. This not only increases cache area and latency, but also requires modifying the access scheduling for the different cache banks to improve synchronization of accesses between different banks. Cache access scheduling is further complicated because sacrificial line store chunks from different data cache lines. Therefore, on a write to a data cache line, Archipelago requires updating the appropriate chunks within the corresponding sacrificial line using an expensive read-modify-write operation. EPT only reorders bits within a word, not across words, and, therefore, avoids all of the above complexities. Finally, EPT can also be applied on top of Archipelago to improve its effectiveness by aggregating faults in unused chunks, as illustrated in Figure 3.14. In this way, EPT is orthogonal to Archipelago.

iPatch [46] protects caches operating at low voltages by reusing unused spaces in other memory structures, such as the store queue, micro-op cache, MSHR buffers, etc., to store redundant copies of words in the cache. IPatch requires modifying a large number of processor components to make them aware of faulty cache words; this may significantly complicate processor de-
sign and verification. In addition, these structures can only protect a small fraction of the cache due to the smaller sizes of these memory structures relative to a cache; as such, IPatch disables unprotected faulty cache words in data lines [46]. Disabling a cache word in a data line leads to uncachable logical words, which can significantly impact performance. EPT, on the other hand, restricts the required modifications to within a cache and does not require disabling cache words in data lines. In addition, EPT can also be applied to a cache along with IPatch and is, therefore, orthogonal to IPatch.

EPT bears some resemblance to bit-interleaving, which statically interleaves adjacent physical bits across different segments of a segmented ECC. For physically adjacent faults, bit-interleaving converts multiple adjacent bits of errors in one segment into single-bit errors in different segments. Causes of physically adjacent bit faults include large alpha particle strikes [47] and complete DRAM chip failures [48]. Faulty SRAM cells at low voltages, however, are randomly distributed across a cache [8], not typically physically adjacent to one another. As such, bit-interleaving does not improve the coverage of faulty SRAM cells during low voltage operation, which EPT does by adaptively modifying the bit ordering in each cache word according to the identified fault pattern of the cache word.

EPT also bears some resemblance to fault dispersion, which seeks to transfer faults from a line with too many faults to lines with fewer faults, in off-chip main memories [49, 50, 51, 52, 53, 54, 55]. In off-chip main memories, a line is typically striped across multiple memory chips/cards called a rank, such that all chips/cards in a rank receive the same memory address input and operate in lockstep to satisfy a single memory request. Exploiting this architecture, prior works perform fault dispersion by physically swapping the memory chips/cards or by reconfiguring the memory chips/cards in a rank to access different intra-chip/card locations for the same address presented to all the chips/cards in the rank. As such, these prior works differ vastly in implementation from EPT, which targets on-chip memories. Fault dispersion in the context of L1 caches equates to transferring logical bits between words in different sets, which requires accessing multiple cache words per access to a faulty cache word; this can incur high latency overheads for low voltage SRAM caches, where latencies are low and fault rates are high, unlike off-chip main memories, where latencies are high and the fault rates are low. Finally, unlike fault dispersion, EPT does not reduce the number of errors
in a cache word, but simply transforms the error pattern generated by the cache word. As such, EPT can also be applied on top of fault dispersion, and is, therefore, also orthogonal to fault dispersion.

3.7 Summary

In this chapter, we presented error pattern transformation, a general technique for low cost error correction which enables memory voltages to be scaled further than prior works on error correction. We observed that although many ECCs only guarantee correction of a small number of errors, they can actually correct a large number of erroneous bits if these bits are in the particular error patterns. Since the same physical fault pattern in a cache word can manifest as different error patterns depending on the ordering of the logical bits stored in a physical cache word, EPT adaptively rearranges the logical bit to physical bit mapping per word according to the known BIST-detectable fault pattern in the physical word. The adaptive logical bit to physical bit mapping transforms many uncorrectable error patterns in the logical words into correctable error patterns and, therefore, improves ECC error coverage and reduces the minimum required voltage of operation. This reduces the minimum voltage at which memory can run by 70mV over the best low-latency ECC baseline leading to a 25.7% core-wide power reduction for an ARM Cortex-A7-like core. Energy per instruction is reduced by 15.7% compared to the best baseline.
Chapter 4

A Scalable Approach to Symbolic Hardware-Software Co-Analysis

Symbolic simulation and symbolic execution techniques have long been used for verifying designs and testing software. However, as this dissertation demonstrates in Chapters 5 to 8, symbolic simulation can be effectively used to reliably reduce power and cost as well as provide security guarantees by characterizing unused hardware resources and tracking information flows across all possible executions of a specific application running on a processor. Like other symbolic simulation and execution techniques, such symbolic hardware-software co-analysis does not scale well to complex applications, due to an explosion in the number of execution paths that must be analyzed to characterize all possible executions of an application. Furthermore, since application-specific power optimizations require guaranteed coverage of all possible execution states of an application, heuristics for increasing the scalability of symbolic simulation cannot be applied, since they improve scalability by sacrificing coverage. In this chapter, we first propose a naive symbolic simulation based hardware-software co-analysis and then propose a technique for performing a symbolic co-analysis that can analyze even complex applications in a scalable fashion while maintaining the conservative guarantees required to enable application-specific power optimizations. Our scalable technique reduces analysis runtime and also enables analysis to complete for applications that were too complex to be analyzed by naive techniques.

4.1 Introduction

Based on the application-specific nature of many emerging ultra-low-power systems, we propose application-specific power and cost reduction techniques that identify hardware resources (e.g., gates) in a processor that cannot be
exercised by the application running on the processor and eliminate power or cost (e.g., area or power supply allocation) required to support those resources. However, such application-specific optimizations can only be safely applied if an analysis technique can guarantee that the application running on the processor will never use the resources for any possible execution of the application, for any inputs. Eliminating power for resources that could be used by the application could lead to incorrect execution of the application. For example, power gating a gate that was incorrectly identified as “unused” but is actually exercised by an application can result in the application producing incorrect outputs or crashing. Given the need for guarantees and the inability to achieve such guarantees through input-based application profiling, our proposed application-specific power and cost reduction techniques rely on a symbolic simulation [56] of the application on the processor hardware to identify hardware resources that are guaranteed to not be used across all possible executions of an application. By propagating symbols that represent unknown logic values for all inputs to an application, it is possible to determine all possible hardware resources that could be used by the application in an input-independent fashion. The input-independent activity profiles generated by such a symbolic simulation of an application running on a processor can be leveraged to identify worst-case timing, power, and energy characteristics for a low-power system and to eliminate power used by resources that the system’s captive application is guaranteed to never use.

The symbolic simulation used for application-specific power management differs from prior work on symbolic simulation in two fundamental ways. First, it is application-specific, meaning that the hardware design is simulated in the context of a particular application binary to identify application-specific power management opportunities that can be exploited independently of how the application is executed. Second, rather than trying to synthesize hardware test vectors to exercise the entire design, it attempts to analyze which gates can possibly be exercised by an application. Symbolic hardware-software co-analysis is also related to symbolic execution [57]; however, it differs in that it considers every possible execution of a given application for all possible inputs, rather than determining a specific set of inputs that will test the application. Also, prior works on symbolic execution do not consider the application-specific effects of software on hardware.

Since the symbolic hardware-software co-analysis used by emerging application-
specific power management techniques must explore all possible execution states of an application on a processor, it suffers from the same scalability limitations characteristic of symbolic simulation and symbolic execution – namely, the state-space explodes for applications with complex control structures, due to a large number of possible execution paths. Many heuristics have been proposed to tame the scalability problems of symbolic simulation and execution [58, 59, 60, 57, 61]; however, existing heuristics for scalable symbolic simulation and symbolic execution cannot be applied to the symbolic co-analysis used for application-specific power management, since existing heuristics sacrifice perfect coverage to improve scalability. Therefore, existing techniques cannot guarantee that all possible inputs and execution states of an application on a processor are considered, and consequently, cannot guarantee that all possible hardware resources (e.g., gates) that an application can exercise in a processor will be identified. As a result, these techniques cannot be used to enable application-specific power management techniques.

So, while this naive symbolic co-analysis for application-specific power management can provide guaranteed coverage of all possible execution states for an application, it cannot scale to analyze applications with complex control structures. While this may not be a problem for many applications used in ultra-low-power systems, since they tend to be simple, more complex control structures that cannot be analyzed by the existing approach to symbolic co-analysis [62] (e.g., input-dependent loops) can be found even in simple embedded applications (see Section 4.5). Analysis of applications with more complex control structures requires a scalable symbolic co-analysis technique that can (a) characterize application behavior for all possible inputs and execution states, (b) allow analysis of applications with arbitrarily complex control flow, and (c) guarantee that all hardware resources that can possibly be exercised by an application will be identified.

In this chapter, we propose a symbolic hardware-software co-analysis technique that can analyze an application with arbitrarily complex control flow and guarantee that no gate marked as untoggled can ever be toggled by any execution of the application. Our technique is based on the observation that many branches within the execution tree have similar state and toggling behavior. Analysis of any branch of the execution tree corresponding to a state that has been previously simulated can be terminated. In addition,
similar states for a branch can be merged into a conservative state, where differing state variables are assumed to have unknown logic values. Analysis can also be terminated for a state that has already been covered by a more conservative version of the state (see example in Section 4.2).

This chapter makes the following contributions:

- We propose a scalable symbolic hardware-software co-analysis technique that enables input-independent analysis for complex applications, even applications with complex control structures such as input-dependent control structures and infinite control structures that cannot be analyzed by existing techniques. The proposed technique guarantees that all gates that can possibly be exercised by an application will be identified.

- We show that the proposed analysis technique significantly reduces analysis time for complex applications, and can even reduce analysis time for applications without complex control flow. Compared to existing techniques, analysis time is reduced by up to 97.2%, and by 41.7% on average.

- We demonstrate that even though our technique is more conservative than previous approaches when analyzing complex applications, the additional gates identified as toggled is small—less than 2.5%.

4.2 Background and Motivating Insight

Since the gates in a processor that are toggled by an application may vary significantly for different inputs, application-specific identification of exercisable gates in a processor must consider all possible inputs to guarantee that all exercisable gates are identified. We propose a co-analysis that achieves this guarantee through symbolic simulation of the application on the processor netlist, where inputs (and uninitialized values) are represented as unknown logic values (Xs), as described in Algorithm 1. To perform co-analysis, all memory cells and gate values are initialized to X, the application binary is loaded into the program memory, and the reset signal is toggled. This starting state of the processor is pushed onto a stack of states that must be simulated, and symbolic simulation begins by popping the initial state off the stack and simulating from that state. Any time an input is read, its bits
are represented by all Xs. Simulation values are propagated as normal, and each simulation state is annotated with a list of exercisable gates observed in that state. If an X ever propagates to the PC, indicating input-dependent control flow, a branch is created in the symbolic execution tree, and states corresponding to all possible next PC values are pushed onto the stack so that all possible paths through the application are analyzed. A depth-first simulation of the application’s control flow graph continues until all execution paths have been explored.

Algorithm 1 Naive Symbolic Co-analysis

1. Procedure GateActivityAnalysis(app_binary, design_netlist)
2. Initialize all memory cells and all gates in design_netlist to X
3. Load app_binary into program memory
4. Propagate reset toggle signal
5. s ← State at start of app_binary
6. Symbolic Execution Tree T.set_root(s)
7. Unprocessed execution points stack, U.push(s)
8. while U ! = ∅ do
9. e ← U.pop()
10. while e.nextPC != X and !e.END do
11. e.setInputsX() // set all peripheral port inputs to Xs
12. e′ ← propagateGateValues(e) // perform simulation for this cycle
13. e.annotateGateActivity(e,e′) // annotate tree point with activity
14. e.addNextState(e′) // add to execution tree
15. e ← e′ // process next cycle
16. end while
17. if e.nextPC == X then
18. for all a ∈ possibleNextPCVals(e) do
19. e′ ← e.updateNextPC(a)
20. U.push(e′)
21. T.insert(e′)
22. end for
23. end if
24. end while

Unfortunately, the co-analysis technique used in prior work cannot analyze applications with complex control flow or infinite loops. Figure 4.1 shows such an application, where an unconditional branch jumps from the end basic block back to the begin basic block, resulting in an infinite loop. Each time the branch (jl) instruction on line 6 is encountered, the else block will be pushed onto the stack (lines 17 and 18 in Algorithm 1), while simulation continues down the then block. However, since the end of the application is never reached, the stack will never be popped, and symbolic simulation will

1Any 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.
Figure 4.1: Symbolic hardware-software co-analysis involves using unknown logic values for application inputs to characterize application-induced processor behavior for all possible executions of an application.

never finish.

A closer look at the application in Figure 4.1 reveals an insight. The program consists of a loop that reads an input, sets the value of either register \( r_4 \) or \( r_5 \), depending on the input value, then subtracts the two registers. Although the loop iterates infinitely, only two possible simulation states exist at instruction 10 (i.e., \( \langle r_4, r_5, r_6 \rangle \) is either \( \langle 1, 0, 1 \rangle \) or \( \langle 0, 1, -1 \rangle \)). These two states only differ in a small fraction of the processor’s state (i.e., \( r_6 \) and one bit each of \( r_4 \) and \( r_5 \)). Thus, a conservative state formed by merging the two states such that differing state variables are set to unknown values (\( X \)s) can represent both states with little loss of toggling information. After simulating a conservative state that represents both possible states, any future simulation of execution paths at line 10 can safely be terminated, since they will not identify any new toggling behavior. In this way, even an application with an infinite execution tree can be analyzed, while still guaranteeing that the maximal set of gates that an application can toggle will be identified.

4.3 Scalable Symbolic Co-Analysis

The symbolic co-analysis technique proposed in prior work on application-specific power management [62] is not scalable for applications with complex
control flow, because it attempts to evaluate gate-level activity for all possible execution paths through an application. In some cases (e.g., infinite loops), evaluation of all possible execution paths is not even possible. The goal of our proposed scalable symbolic co-analysis technique is to conservatively avoid the exploration of paths (states) for which the worst-case toggle behavior has already been simulated and to reduce the number of states that must be explored by forming and simulating conservative states that cover many possible execution paths.

Algorithm 2 Scalable Symbolic Co-analysis

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

Algorithm 2 describes the proposed scalable symbolic co-analysis technique. The analysis initializes the symbolic simulation in the same way as Algorithm 1 (i.e., all nets and memory cells are initialized to X, the PC is loaded with the application’s first address, and reset is toggled), with one addition – Algorithm 2 also initializes a conservative system state map. This map holds the conservative simulation values for each net and memory
location in the design at any instruction that alters the PC (besides incrementing). Only the states at PC-altering instructions (e.g., branch, jump, etc.) are stored, since they are the instructions that can cause path explosion due to input-dependent or infinite control structures. Each map entry’s key is the PC value of the PC-altering instruction (e.g., a branch’s address in program memory). The conservative value stored in the map for a state variable is assumed to be unknown ($X$) for any net that has been observed to have different values (i.e., ‘0’ and ‘1’) in a state with the same PC. Assigning a value of $X$ to a net carries the assumption that the net can be toggled by the application for some input assignment.

Once initialization is complete, simulation begins, continuing until all paths have been explored or have been determined to be covered by a previously simulated state. During each visit to a PC-altering instruction, the current state is compared with the conservative state stored for that instruction (PC). If the current state is a substate of the stored state (i.e., the states are identical OR the stored state has $X$s in all state variables where the states differ), then all paths through the current state have already been analyzed in a previous portion of the symbolic simulation, and the current execution path can be safely terminated. If the current state is not a substate of the stored state, a new conservative symbolic state is generated by assigning any nets that differ in value between the current state and the stored conservative state to $X$s.\footnote{The reason an $X$ produces the worst-case toggling behavior is that even if a net has an $X$ value in two consecutive cycles, the analysis tool considers it a possible toggle.} This new conservative state is loaded as the processor state before continuing symbolic simulation and is also stored into the conservative state map in place of the previous stored state. Symbolic simulation must continue to explore this execution path from the new conservative state, because it includes new toggled gates, and therefore, the worst-case toggling activity may not have been observed yet. Symbolic simulation then continues as described by Algorithm 2.

Figure 4.2 shows an example of the proposed technique analyzing an application (from Figure 4.1) that Algorithm 1 is unable to analyze. The application contains an infinite loop of four basic blocks; the execution of two of the basic blocks is dependent on the input value read on line 4. Algorithm 1 would attempt to explore the entire infinite execution tree (including grayed-out blocks and beyond). However, inspecting the code, it is clear that
there are two possible system states at the end of one iteration of the loop – $(r4, r5, r6)$ is either $(1, 0, 1)$ or $(0, 1, -1)$. Therefore, continuing to explore the execution tree after these two states have been observed will not uncover any new toggling behavior.

During scalable symbolic co-analysis, when the conditional branch instruction at line 6 is first reached, state $S_0$ is added to the conservative system state map with a key of 6 (the branch’s PC). At this point $r4$, $r5$, and $r6$ are 0, while $r15$ is all Xs (because its value is read from an input). As symbolic simulation continues down the then path, entries $S_1$ and $S_2$ are added for the branches at lines 8 and 11, respectively. When symbolic simulation reaches the branch at line 6 again, $r6$ has a value of 1, which requires a new conservative state, $S_3$, to replace $S_0$, where $r6$’s value is 0...0X, because the least significant bit of $r6$ was observed once as a 0 and once as a 1 during the branch instruction at line 6. Continuing, state $S_1$ must be replaced by state $S_4$, because the simulation value of $r6$ is 0...0X during the next simulation of the branch at line 8. Since the value of $r6$ is then overwritten to be 1 again prior to line 11, this execution path observes state $S_2$ for a second time, indicating that no further exploration is required, and the path is terminated. Next, the last else block is popped off the execution points stack. For this path, $r6$ becomes -1 (i.e., 1...11), which is not a substate of the latest conservative state stored for the branch at line 11 – $S_2$. Thus, a new state, $S_5$, replaces $S_2$. At the end of the next iteration of the main loop, the current state is a substate of $S_5$, so exploration is halted for that path and the top
of the execution points stack is popped. After this, all states encountered are substates of previously explored states stored in the conservative system state map. In this example, our scalable symbolic co-analysis technique only simulates 35 dynamic instructions, rather than the infinite number the previous co-analysis technique would attempt to simulate. The proposed scalable symbolic co-analysis technique captures the worst-case toggling activity of an application with $O(n^2 \times m)$ complexity, where $n$ is the number of basic blocks in the application and $m$ is the number of gates in the processor’s netlist.

One drawback of the proposed scalable technique is the inaccuracy introduced by being conservative when recording toggling activity. Consider Figure 4.3. Assume that states $S_0$ and $S_1$ are two different states observed for the same PC, arrived at sequentially by the baseline co-analysis. In the proposed scalable technique, when simulation arrives at state $S_1$ after having previously observed $S_0$ for the same PC, a new conservative state, Cons$S_1$, will be created and stored as the current conservative state for this PC. Cons$S_1$ is then used to continue symbolic simulation instead of $S_1$. Therefore, the scalable approach must conservatively assume that both gates $a$ and $b$ are toggling, when the baseline co-analysis may show them to not toggle. Since usually only a small number of state values differ between executions of the same static instruction, the difference between the scalable and baseline symbolic co-analysis approaches is expected to be small. Our evaluations in Section 4.5 confirm that this is the case. Also, note that the inaccuracy introduced by our scalable technique will only make analysis more conservative. This means that application-specific power management techniques enabled by symbolic co-analysis may have somewhat lower benefits but will always guarantee correctness.

4.4 Methodology

We perform evaluations on a silicon-proven processor – openMSP430 [63] – synthesized, placed and routed in TSMC 65GP (65nm) technology using Synopsys Design Compiler and Cadence EDI System. The processor was implemented for an operating point of 1V and 100MHz. We implemented scalable symbolic co-analysis in a custom gate-level simulator that was built
Figure 4.3: Simulating from a conservative state in our scalable symbolic co-analysis technique may identify some gates as possibly exercisable that would not be reported as such by the baseline symbolic co-analysis approach. The conservative nature of our approach allows analysis of applications with complex control structures while maintaining the guarantee that all exercisable gates will be identified.

Table 4.1: Benchmarks.

<table>
<thead>
<tr>
<th>Embedded Sensor Benchmarks [64]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult, binSearch, tea8, intFilt, div, inSort, rle, tHold, intAVG</td>
</tr>
<tr>
<td>EEMBC Embedded Benchmarks [65]</td>
</tr>
<tr>
<td>Autocorr, ConvEn, FFT, Viterbi</td>
</tr>
<tr>
<td>Recursive Benchmark</td>
</tr>
<tr>
<td>MergeSort</td>
</tr>
</tbody>
</table>

in-house. We show results for all benchmarks from [64], all EEMBC benchmarks that fit in the program memory of our processor, as well as a recursive benchmark designed to stress-test the scalability of our symbolic hardware-software co-analysis technique with a complex control structure not found in the rest of our benchmarks (Table 4.1). Experiments are performed on a server housing two Intel Xeon E-2640 processors (eight cores each, 2GHz operating frequency, 64GB RAM).

4.5 Results

To illustrate the benefits of our scalable technique for symbolic co-analysis, we compare the runtime of the baseline symbolic co-analysis technique (Algorithm 1) against our scalable technique (Algorithm 2). Table 4.2 presents a comparison of analysis times for the benchmark applications. Due to complex or infinite control structures, the baseline technique did not finish for some of the benchmarks (div, inSort, intAVG, rle, Viterbi), even after 15 hours of simulation. For these benchmarks, we show the analysis time as $\infty$. To supplement our analysis, we implemented smaller versions of these
Table 4.2: Scalable symbolic co-analysis reduces analysis time compared to baseline symbolic co-analysis and enables analysis of applications with complex control structures.\(^3\)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Analysis Time (s)</th>
<th>Baseline</th>
<th>Proposed</th>
<th>%Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>715</td>
<td>23</td>
<td>97%</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>inSort</td>
<td>∞</td>
<td>25</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>inSort_small</td>
<td>459</td>
<td>13</td>
<td>97%</td>
<td></td>
</tr>
<tr>
<td>intAVG</td>
<td>∞</td>
<td>116</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>intFilt</td>
<td>1378</td>
<td>1365</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>3</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>rle</td>
<td>∞</td>
<td>20</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>rle_small</td>
<td>187</td>
<td>6</td>
<td>97%</td>
<td></td>
</tr>
<tr>
<td>tHold</td>
<td>7</td>
<td>7</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>tea8</td>
<td>20</td>
<td>20</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>10542</td>
<td>10498</td>
<td>0.4%</td>
<td></td>
</tr>
<tr>
<td>AutoCorr</td>
<td>91</td>
<td>50</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>convEn</td>
<td>1722</td>
<td>1401</td>
<td>19%</td>
<td></td>
</tr>
<tr>
<td>Viterbi</td>
<td>∞</td>
<td>443</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>MergeSort</td>
<td>3112</td>
<td>3007</td>
<td>3%</td>
<td></td>
</tr>
</tbody>
</table>

benchmarks (denoted by _small) that either read fewer inputs or read input values that are bounded by a small value. Even the smaller versions of div and intAVG did not finish after simulating for 15 hours. This is because div performs an XOR of an input with itself to determine the number of iterations required to finish the division operation. Thus, even a single X in the input can propagate to the computed iteration bound, leading to an infinite loop. Since intAVG uses a division operation in its computation, it suffers the same fate. The inSort and rle applications do not suffer from this issue, and their smaller versions shrink the execution tree such that the baseline can complete analysis; however, scalable symbolic co-analysis achieves significant speedups for these benchmarks, even in their smaller versions, by eliminating redundant analysis of previously explored states.

For some benchmarks, our technique provides little or no benefit over the baseline (e.g., mult, intFilt, tea8, FFT, AutoCorr, convEn). This is because these applications, like many applications for ultra-low-power systems, have simple control flow (e.g., no input-dependent branches or infinite loops), and hence, our scalability techniques are not invoked. While tHold has input-dependent branches, its simple CFG did not present enough opportunities for our technique to have an advantage over the baseline.

For most benchmarks with input-dependent branches (e.g., binSearch, \(^3\)∞ means that the analysis did not complete in 15 hours. _small indicates a benchmark run with a smaller configuration so that the baseline analysis can complete.
Table 4.3: Use of conservative states to improve scalability identifies more gates as possibly exercisable; however, the difference between baseline and scalable symbolic co-analysis is small.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Percentage Gates Toggled</th>
<th>Baseline</th>
<th>Proposed</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>73.17</td>
<td>73.80</td>
<td>0.86%</td>
<td></td>
</tr>
<tr>
<td>div</td>
<td>73.56</td>
<td>74.45</td>
<td>1.20%</td>
<td></td>
</tr>
<tr>
<td>inSort</td>
<td>73.81</td>
<td>74.03</td>
<td>0.29%</td>
<td></td>
</tr>
<tr>
<td>inSort_small</td>
<td>72.72</td>
<td>72.72</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>intAVG</td>
<td>73.87</td>
<td>74.68</td>
<td>1.09%</td>
<td></td>
</tr>
<tr>
<td>intFilt</td>
<td>79.27</td>
<td>79.27</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>78.24</td>
<td>78.24</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>rle</td>
<td>74.82</td>
<td>75.63</td>
<td>1.08%</td>
<td></td>
</tr>
<tr>
<td>rle_small</td>
<td>73.71</td>
<td>73.74</td>
<td>0.04%</td>
<td></td>
</tr>
<tr>
<td>tHold</td>
<td>72.18</td>
<td>72.19</td>
<td>0.01%</td>
<td></td>
</tr>
<tr>
<td>tea8</td>
<td>74.88</td>
<td>74.88</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>80.35</td>
<td>80.35</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>AutoCorr</td>
<td>80.13</td>
<td>80.13</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>convEn</td>
<td>74.96</td>
<td>74.96</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>Viterbi</td>
<td>73.98</td>
<td>74.91</td>
<td>1.01%</td>
<td></td>
</tr>
<tr>
<td>MergeSort</td>
<td>50.36</td>
<td>51.61</td>
<td>2.48%</td>
<td></td>
</tr>
</tbody>
</table>

div, inSort, inSort_small, intAVG, rle, rle_small), the benefits of scalable co-analysis are significant. In fact, the proposed technique is not only significantly faster than the baseline for these benchmarks, but it also makes analysis possible for benchmarks that could not be analyzed by the baseline technique. For the subset of benchmarks that can be analyzed by the baseline, our scalable technique reduces analysis time by up to 97.2%, and by 41.7% on average.

As discussed in Section 4.3, our technique performs faster analysis than the baseline for benchmarks that contain complex control flow, because every time we visit a branch in the control flow graph, we build a conservative state of the system that covers all observed states during visits to the same branch. If, during a visit, the state is a substate of a previously explored state, execution down that branch path can be terminated. Since the new state at a branch always includes all the previously visited states and is bounded by the state where the entire system is marked with $X$s, we are guaranteed to reach a simulation state in which simulation from the branch is no longer necessary.

Although it improves the scalability of symbolic co-analysis, our technique may introduce inaccuracies, since a conservative state may encompass one or more states that an application cannot actually exercise. This can result

---

4A common example is a loop counter turning to all $X$s, which takes at most 16 visits for the openMSP430.
in our technique conservatively identifying more gates as exercisable, compared to the baseline. To compare the accuracy of our co-analysis technique against the baseline, we determine the fraction of gates in the design identified as exercisable by a symbolic co-analysis technique (this metric is used to characterize the number of gates that our scalable but conservative analysis marks as exercisable that are not actually exercisable by the application).

Table 4.3 shows the fraction of gates in the design that are identified as exercisable by each symbolic co-analysis technique. For benchmarks for which the baseline did not finish (div, inSort, intAVG, rle, Viterbi), the table presents the fraction of gates that were toggled until the simulator timed out. For several benchmarks (e.g., intFilt, mult, tHold, tea8, inSort_small), our conservative technique toggles the same number of gates as the baseline. This can be expected for applications like intFilt, mult, and tea8, since they do not have input-dependent branches. In the case of tHold, we observed that the only bits that change during every visit to an input-dependent branch corresponded to the loop counter and the output variable that represents a count of the number of input values that are above a threshold. Since these states, and hence their corresponding gates, would eventually be exercised by the full simulation of the application anyway, the number of toggled gates did not change for tHold. In the case of inSort_small, we shrunk the input vector (to be sorted) to eight entries. Since the loop iterator’s maximum value of 7 (111 in binary) is the largest state reachable in three bits, the baseline can toggle all the exercisable bits during the full execution of the application, and hence, setting the three least significant bits of the iterator to Xs did not introduce any unreachable states. However, we notice that for the actual benchmark inSort, the scalable technique actually marks more gates as exercisable than the baseline. This can be attributed to two factors – (1) the baseline symbolic simulation was unable to finish analyzing the application, and (2) we used an input vector of size 17 (binary representation 10001), which means that the iterator’s conservative representation (XXXXX) encompasses several unreachable states (10010-11111). However, the percentage of extra gates that were identified as exercisable is still small for inSort.

Intuitively, it makes sense that our conservative analysis only increases the number of gates identified as exercisable by a small amount, because even though there are many new potentially infeasible execution paths to
analyze, they are explored on the same piece of hardware, and hence the majority of the gates being exercised during exploration of these paths are expected to be the same. This also applies to several other benchmarks, such as binSearch, div, inSort, and rle, which have input-dependent branches. For these benchmarks, our scalable technique identifies more exercisable gates than the baseline, since it introduces infeasible states, but the number of additional gates is small, for the same reason as explained above.

For the benchmarks where inaccuracy is introduced by conservative states, the maximum inaccuracy in the number of toggled gates from our scalable technique is 1.2%. Considering all the benchmarks, the average percentage degradation in accuracy is 0.5%.

Since the goal of our approach to symbolic hardware-software co-analysis is to enable scalable analysis for applications with arbitrarily complex control structures, in addition to evaluating domain-relevant applications for ultra-low-power embedded systems, we also evaluated a recursive application intended to stress-test the scalability of our symbolic co-analysis technique for a complex control structure. The MergeSort application uses recursion, which can be a stress test for our symbolic co-analysis technique, since the depth of recursion for an application is often input-dependent. Thus, conservative state is invoked to cover the possible execution states of a recursive application. Also, as conservative states are evaluated, the stack used to support recursive function calls becomes more conservative, since multiple calls to the same recursive function would likely push different states onto the stack at the same branch in the control flow graph. Thus, we evaluate MergeSort to verify that (a) our scalable symbolic co-analysis technique can handle recursion and (b) our conservative symbolic co-analysis approach does not introduce too much pessimism due to the interaction of conservative states in a recursive control flow graph.

The results in Table 4.2 show that our scalable symbolic co-analysis technique is able to handle recursion, even generating a slight speedup over the baseline technique. The real test, however, is whether recursion can be handled without introducing too much pessimism. Table 4.3 shows that the pessimism introduced by using conservative state to analyze recursion is negligible. Compared to the baseline, scalable symbolic co-analysis only identifies 2% additional gates as exercisable. Thus, our scalable co-analysis technique is able to analyze an application containing recursion without sig-
Table 4.4: Microarchitectural features in recent embedded processors.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Branch Predictor</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Cortex-M0</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ARM Cortex-M3</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Atmel ATxmega128A4</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Freescale/NXP MC13224v</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Intel Quark-D1000</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Jennic/NXP JN5169</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>SiLab Si2012</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>TI MSP430</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Significantly degrading the potential for application-specific power savings enabled by symbolic hardware-software co-analysis.

4.6 Generality and Limitations

We target our co-analysis for applications with ultra-low power constraints. Low-power processors are already the most widely used type of processor and are also expected to power a large number of emerging applications [66, 67, 68, 69, 70]. Such processors also tend to be simple, run relatively simple applications, and do not support non-determinism (no branch prediction and caching; for example, see Table 4.4). This makes our symbolic simulation-based technique a good fit for such processors. Below, we discuss how our technique may scale for complex processors and applications, if necessary.

More complex processors contain more performance-enhancing features such as large caches, prediction or speculation mechanisms, and out-of-order execution, that introduce non-determinism into the instruction stream. Co-analysis is capable of handling this added non-determinism at the expense of analysis tool runtime or accuracy (i.e., becoming more conservative). For example, by injecting an X as the result of a tag check, both the cache hit and miss paths will be explored in the memory hierarchy. Similarly, since co-analysis already explores taken and not-taken paths for input-dependent branches, it can be adapted to handle branch prediction. In an out-of-order processor, instruction ordering is based on the dependence pattern between instructions. While instructions may execute in different orders depending on the state of pipelines and schedulers, a processor that starts from a known reset state and executes the same piece of code will transition through the
same sequence of states each time. Thus, modifying input-independent CFG exploration to perform input-independent exploration of the data flow graph (DFG) may allow analysis to be extended to out-of-order execution.

For complex applications, CFG complexity increases. This may not be an issue for simple in-order processors (e.g., the ultra-low-power processors studied in this chapter), since the maximum length of instruction sequences (CFG paths) that must be considered is limited based on the number of instructions that can be resident in the processor pipeline at once. However, for complex applications running on complex processors, heuristic techniques may have to be used to improve scalability. While heuristics have been applied to improve scalability in other contexts (e.g., verification) [71, 72], heuristics for our hardware-software co-analysis must be conservative to guarantee that no gate is marked as untoggled when it could be toggled.

In a multi-programmed setting (including systems that support dynamic linking), we take the union of the toggle activities of all applications (caller, callee, and the relevant OS code in case of dynamic linking) to get a conservative toggling activity. For optimizations which require specific timing information, all possible interleavings of such programs must be considered. For self-modifying code, all possible modifications are considered (i.e., all possible executions of the self-modifying code). In the case of fine-grained multi-threading, any state that is not maintained as part of a thread’s context is assumed to have a value of $X$ when symbolic execution is performed for an instruction belonging to the thread. This leads to a safe guarantee of toggling activity for the thread, irrespective of the behavior of the other threads.

Our technique naturally handles state machines that run synchronously with the microcontroller. For state machines that run asynchronously (e.g., ADCs, DACs, bus controllers), we assume the source of non-determinism (e.g., external initialization of a DMA) is an $X$ at all times. This produces the worst-case toggling activity throughout the execution of the software on the processor. Asynchronous state machines are generally much smaller than the actual processor, allowing us to not be overly conservative.

A similar approach can be used to handle interrupts. The effect of an asynchronous interrupt can be characterized by forcing the interrupt pin to always read an $X$. Since this can potentially cause the PC to be updated with an $X$ if the interrupt is enabled, we must also consider each instruction to be a
point at which the PC can be modified and add the interrupt’s service routine to the set of next possible program addresses. Interrupt service routines (ISRs) are regular software routines and can be analyzed with the rest of the code.

Our evaluations in this chapter have been performed in the context of bare-metal design (no OS). While many low-power microprocessors and a large segment of embedded systems are bare-metal systems (application running on the processor without an operating system (OS)) [73, 74, 75, 76], use of an OS is common in several embedded application domains, as well as in more complex systems. In such systems, system code must be analyzed in addition to application code to identify power gating opportunities. Our symbolic analysis has been applied to OSes such as FreeRTOS [77] (see Chapters 6 and 8).

4.7 Related Work

Our work on scalable symbolic hardware-software co-analysis is related to work on symbolic execution and symbolic simulation. Symbolic execution involves running a program using symbolic inputs instead of regular inputs, for program analysis and test generation, while symbolic simulation involves simulating a hardware design using symbolic inputs for design analysis, verification, and test generation.

Symbolic execution has been studied extensively in the software community [57]. Since the goal of symbolic execution is to either analyze an application or generate test inputs, the search space is the entire execution tree of the application, which can be exponentially large or even infinite. Hence, several heuristics have been developed to deal with the path explosion problem [57, 61]. However, these techniques are only applicable at the software level and do not allow analysis of the behavior of a processor running the application. They also cannot be used when conservative guarantees are required, since they sacrifice coverage to improve scalability.

Symbolic simulation is a technique used in the hardware community to analyze a hardware design for logic and timing verification and sequential test generation [56]. Similar to the path explosion problem of symbolic execution, symbolic simulation techniques suffer from the state explosion problem.
for which several BDD-based and SAT-based techniques have been developed [58, 59, 60]. Since the goal of symbolic simulation techniques is to analyze a hardware design for verification and test generation, the search space is the entire state space of the processor.

Symbolic simulation has been applied in other contexts, such as worst-case execution time analysis [78, 79] and worst-case energy analysis [80]. These techniques scale symbolic simulation for their respective domains. No technique exists that scales symbolic analysis of an application binary on the netlist of a processor.

4.8 Summary

Symbolic simulation of an application binary on the gate-level netlist of a processor can be used for application-specific power, energy, area, and security optimizations, if the simulation can guarantee that all possible execution states of the application are explored. Unfortunately, symbolic simulation that must explore all possible execution states does not scale well for complex applications, and existing heuristic for improving scalability void coverage guarantees. In this chapter, we proposed a scalable symbolic hardware-software co-analysis technique that guarantees that all possible gates that an application can exercise in a processor will be identified. We showed that our technique can reduce application analysis time and can even enable analysis of applications with complex control structures that cannot be analyzed by naive techniques.
Chapter 5

Enabling Effective Module-Oblivious Power Gating for Embedded Processors

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

5.1 Introduction

A large number of existing and emerging computing applications require ultra-low-power operation and extreme energy efficiency [68, 66, 70, 69, 67,
Notable among these are the Internet of Things, sensor networks, wearables, and health monitors. The 2015 ITRS report projects power and energy constraints of these systems to be even tighter in the future [82]. Unsurprisingly, these applications rely on low-power microcontrollers and microprocessors that have become the most widely used type of processor in production today [2, 3, 4].

The ultra-low power and energy requirements of emerging applications, along with the increasing leakage energy dissipation that has accompanied CMOS scaling [83], have fueled interest in aggressive power gating techniques. Conventional aggressive power gating techniques perform module-based power gating, i.e., power gating of RTL modules during periods of inactivity [84, 85, 86]. An RTL module is encapsulated with a well-defined port list, making it relatively easy to determine when a module is inactive based on input signals in the port list.

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

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

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

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

Our proposed technique is automated, requires no programmer intervention, and incurs low management overhead. Also, while the technique is general, it is best suited for embedded systems. Embedded system designers routinely perform hardware/software co-design [87, 88] or license hardware IP [89, 90], so they often have access to both RTL and software binary – the inputs needed by our power gating framework. Also, embedded processors and applications tend to be simple, so our symbolic simulation-based analysis scales well in such settings.

This chapter makes the following contributions.

1Power gating binary annotation can be offered as a cloud compilation service by the hardware system vendor in non-embedded settings, where the application developer does not have access to the processor description [91, 92, 93].
Table 5.1: Power domains in recent processors.

<table>
<thead>
<tr>
<th>Processor</th>
<th>#domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI MSP430 Wolverine</td>
<td>“many” [94]</td>
</tr>
<tr>
<td>ARM Cortex-A9</td>
<td>14 [95]</td>
</tr>
<tr>
<td>ARM Cortex-A15</td>
<td>8 [96]</td>
</tr>
<tr>
<td>Atmel SAML21</td>
<td>5 [97]</td>
</tr>
<tr>
<td>Intel Atom E6</td>
<td>15+ [98, 99]</td>
</tr>
</tbody>
</table>

- We make a case for module-oblivious power gating. We show that module-oblivious power gating can result in 2× higher leakage energy savings compared to state-of-the-art module-based power gating.
- To enable module-oblivious power gating, we present a fully automated technique that performs co-analysis of an embedded system’s processor netlist and application binary to make safe, aggressive power gating decisions.2 To the best of our knowledge, this is the first technique for module-oblivious power gating.
- We fully implement module-oblivious and module-based power gating in openMSP430 using an industry-standard UPF methodology that accounts for all power gating overheads. We demonstrate that module-oblivious power gating can achieve 2× higher leakage energy savings compared to module-based power gating. We show that module-oblivious gating based on our techniques achieves leakage energy savings that are within 8% of optimal.
- Finally, we show that our technique for managing module-oblivious domains is effective even at managing conventional module-based domains. It saves 12% more energy than an idealized implementation of Idle Count—a hardware-based domain management technique for module-based domains. Our benefits are within 6% of optimal for module-based domains.

5.2 Related Work

While a large body of work exists on processor- and core-level power gating [84, 85, 100, 101, 102, 103], emerging power- and energy-constrained applications have fueled recent work on aggressive module-based power gating techniques [86, 102, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114].

---

2Our automated co-analysis tool for module-oblivious power gating is available for download at the following link: http://www.removed.for.blind.review
These techniques, including those that adaptively re-size microarchitectural structures [86, 85], and techniques that target uncore components (e.g., on-chip routers [115, 116, 117]), focus on power gating of RTL modules. Since module-based domains are smaller and more homogeneous, they may provide more frequent opportunities for aggressive power gating than processor- or core-level solutions. Table 5.1 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.

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

In terms of power domain management, prior works have used software- or hardware-based management. Software-based domain management techniques [118, 86] infer when power domains will be inactive by analyzing an application binary. This requires the functionality of managed power domains to be software-visible. Prior techniques for software-based power domain management cannot be used for module-oblivious power domains, because such domains may contain logic that belongs to many modules and contributes to many fine-grained functionalities, making it impossible to infer activity of module-oblivious domains from software alone.

Hardware-based domain management techniques use hardware monitors to detect when power domains are inactive [100, 85, 101, 114, 108, 109]. Such an approach is feasible for prior works on aggressive module-based power gating, because an RTL module is encapsulated with a well-defined interface (port list) and function. Thus, it is possible to infer domain activity from a relatively small number of signals. Hardware monitoring is infeasible for module-oblivious domains, however, because they do not have a well-defined interface or functionality. As such, the number of signals that must be monitored to infer domain activity is prohibitively large.
5.3 Motivation

5.3.1 A Case for Module-Oblivious Power Domains for Microprocessors

There are several reasons why module-oblivious power domains may provide significantly more opportunities for power gating than module-based domains in microprocessors. One reason is that logic in microarchitectural modules is grouped together largely based on functionality or position in the processor pipeline, which does not necessarily imply correlation in terms of activity. It may often be the case that different logic partitions within the same microarchitectural module have very different activity profiles. For example, many microarchitectural modules support “one-hot” logic. This implies that each logical state is mutually exclusive of all other states. Similarly, each instruction selects and executes on one execution unit. This leaves all other execution units idle. Furthermore, it is common for several modules to have parts that are nearly always active and other parts that are nearly always idle. This weak or anti-correlation between the activity profiles of different parts within a module limits the effectiveness of power gating for module-based domains. Figure 5.1a shows activity profiles for the frontend and execution unit modules of an openMSP430 processor [63] running an encryption application (tea8), where each module has been divided into two sub-modules. In the figure, a high/low value indicates that a sub-module is active/idle. Since both the frontend and the execution unit have at least one part active during nearly every instant of this time period, there is no opportunity to power gate either module. Stated differently, \( fnd_A \) and \( exu_A \) prevent the frontend and execution unit from being power gated, even though \( fnd_B \) and \( exu_B \) are almost completely inactive. If, however, \( fnd_A \) and \( exu_A \) were combined to form one power domain and \( fnd_B \) and \( exu_B \) formed a second domain, the second domain could be power gated during this time period. Uncorrelated activity within modules and correlated activity across modules indicates that there may be significant opportunities to perform more aggressive power gating with module-oblivious power domains.

Another reason for correlated activity across module boundaries is that logic in one module often drives logic in another module. Although the entire modules are unlikely to have correlated activity, the driving and driven parts
(a) Uncorrelated activity within a module can prevent power gating of module-based domains, whereas module-oblivious domains allow more aggressive power gating.

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

Figure 5.1: Activity profiles for different module partitions suggests that module-oblivious domains may provide significantly more opportunities for power gating than module-based domains.
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 5.1b illustrates this behavior with an example for an application that performs multiplication, where the multiplier and memory backbone modules of the openMSP430 processor have each been divided into two sub-modules. Sub-module $mbb_A$ contains the peripheral data input bus that feeds input data to the multiplier (since the multiplier is one of the peripherals). The activity of this sub-module is highly correlated to that of $mul_B$, which contains the input side of the multiplier. When domain wakeup overhead is considered, module-based power domains do not allow any power gating of these modules (blue activity profiles). However, when module-oblivious domains are formed (red activity profiles), both module-oblivious domains can be power gated for significant portions of this time period.

Figure 5.2 is a correlation matrix that shows the correlation between each pair of sub-modules in the openMSP430 processor, where each module has been partitioned into four sub-modules and correlation equals the fraction of cycles in which two sub-modules exhibit the same activity (active or idle). The dashed boxes along the main diagonal encircle correlation scores for sub-modules that belong to the same module. It can be observed that not all parts of a module have correlated activity, and in many cases, different parts of the same module have highly uncorrelated activity. Tracing down a row corresponding to a given sub-module, it can be observed that there always exist one or more sub-modules from different modules that have more correlated activity profiles than a sub-module from the same module. For example, in the row showing correlations for the last sub-module in the frontend, we have encircled all the (10) sub-modules from different modules that are more strongly correlated to this frontend sub-module than any of the other frontend sub-modules. These observations suggest that module-based power domains may often miss opportunities to power gate idle logic, whereas module-oblivious power domains may provide significantly more opportunities to power gate larger areas of logic for longer periods of time. In Section 5.6, we show that a full-fledged UPF implementation of openMSP430 with module-oblivious domains achieves up to $2 \times$ more leakage savings than an implementation with module-based domains.
5.3.2 A Case for a Novel Management Technique for Module-Oblivious Domains

Reaping the power benefits enabled by module-oblivious domains requires a power domain management technique that can determine when domains are idle / active and power them off / on accordingly. Unfortunately, existing techniques that manage module-based domains through software or hardware cannot be used for module-oblivious domains. Consider existing software-based management techniques. Software-based management is possible when domain activity can be inferred from software, as is the case for many module-based domains [84]. In the example code listing in Figure 5.3, domain D0 is a module-based domain corresponding to the adder in the execution unit. Since the adder module has a well-defined architectural function, it is possible to infer when the domain must be powered on. For example, instructions 4 (compare) and 9 (subtraction) use the adder, so domain D0 must be powered on when those instructions reach the execution stage. The adder can potentially be powered off for other instructions, since they do not use the adder.

For a module-oblivious domain, however, it is not possible to infer domain
activity from software alone. A module-oblivious domain does not have a well-defined architectural function. It is a collection of gates with correlated activity profiles that may belong to many modules and contribute to many functionalities. For example, domain D3 in Figure 5.3 corresponds to the module-oblivious domain in Figure 5.4c (see Section 5.5.1 for details of how module-oblivious domains are constructed). The domain contains gates from ten different modules, including glue logic, the memory backbone, and clock generation logic for which activity cannot be inferred based on software.

Similarly, existing hardware-based domain management techniques are infeasible for module-oblivious domains. Hardware-based domain management dynamically determines when a power domain is idle / active based on processor control signals. This can be relatively straightforward for module-based designs, since RTL modules are encapsulated, with a well-defined interface (port list) and functional description. For example, consider D0 in Figure 5.3 – the adder module. To determine if this domain will be active, hardware-based management logic only needs to detect if a decoded opcode corresponds to one of the instructions that uses the adder. Figure 5.5a shows the verilog statements that can be added to the decode stage to infer the activity of the adder. Synthesized, this logic corresponds to only six gates.

On the other hand, domain management logic for a module-oblivious domain is not simple. Since module-oblivious domains are not nicely encapsu-
Figure 5.4: Breakdown of domain composition for each module-oblivious power domain. All four domains have gates from at least eight microarchitectural modules.

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

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

(b) Verilog statements for inferring the activity of the module-oblivious domain from Figure 5.4c – synthesizes to 4010 gates.

module domain_activity_detector_D3 (
    in, // domain inputs
    ff_d, // domain ff D-pins
    ff_q, // domain ff Q-pins
    clk, wake_up_domain);
input [704:0] in; input [648:0] ff_d;
input [648:0] ff_q; input clk;
output wake_up_domain; reg [704:0] in_delay;
always @ (posedge clk)
begin
    in_delay <= in;
end
wire [704:0] in_toggled = in ^ in_delay;
wire [648:0] ff_toggled = ff_d ^ ff_q;
wire any_input_toggled = (in_toggled);
wire any_ff_toggled = (ff_toggled);
wire wake_up_domain = (any_input_toggled | any_ff_toggled);
endmodule

Figure 5.5: Hardware-based domain management logic for a module-based domain can be relatively simple, whereas domain management logic for a module-oblivious domain is prohibitively expensive.
Table 5.2: Overheads for hardware-based management of module-oblivious power domains in openMSP430.

<table>
<thead>
<tr>
<th>Domains</th>
<th>Gate Count</th>
<th>FF Count</th>
<th>Domain Inputs</th>
<th>Area Overhead</th>
<th>Static Power Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6695</td>
<td>784</td>
<td>947</td>
<td>143.36%</td>
<td>136.44%</td>
</tr>
<tr>
<td>3</td>
<td>1203</td>
<td></td>
<td>1203</td>
<td>159.38%</td>
<td>151.30%</td>
</tr>
<tr>
<td>4</td>
<td>1450</td>
<td></td>
<td>1450</td>
<td>183.52%</td>
<td>173.69%</td>
</tr>
</tbody>
</table>

lated with a well-defined interface and function, the only way to infer their activity in hardware is to monitor activity on all input nets that cross the domain boundary. Additionally, state elements (flip-flops) inside the domain must be monitored for activity, since a state machine inside the domain could be active even without triggering any activity at the domain boundary. Figure 5.5b shows the verilog statements that infer the activity of the module-oblivious domain D3 from Figure 5.4c. Synthesized, this logic corresponds to 4010 gates.

The overhead of managing module-oblivious domains in hardware becomes prohibitive when the full processor is considered. Table 5.2 shows the area overhead incurred by hardware-based domain management logic in openMSP430 for two-, three-, and four-domain designs. The overheads preclude any possible benefits from aggressive power gating, prohibiting the use of hardware-based domain management for module-oblivious power gating.

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

5.4 A Co-Analysis Based Approach for Module-Oblivious Power Gating

A power domain management technique must infer domain activity to determine when domains can be powered off, while guaranteeing that they will be powered on when active. Since a module-oblivious domain may contain an arbitrary set of gates, inferring domain activity requires gate-level analysis
of software execution on a processor. Activity analysis cannot be based on profiling (i.e., observing activity for several benchmark runs with different input sets), since profiling is input-specific and may result in incorrect management decisions when in-field inputs are different than the inputs characterized during profiling. An incorrect management decision is unacceptable, since it may lead to incorrect program execution (e.g., when a domain needed by the program is turned off). Below, we describe a novel approach that uses symbolic simulation to characterize the gate-level activity of an application on a processor to generate power gating decisions for module-oblivious power domains. The symbolic simulation uses unknown logic values (Xs) for all inputs so that the generated activity profile characterizes all possible executions of the application for all possible inputs. We use the results of input-independent activity analysis to generate instruction-level power domain management decisions that achieve near-optimal power benefits while guaranteeing that all domains are powered on whenever needed. Figure 5.6 provides an overview of our module-oblivious power gating technique.

5.4.1 Gate Activity Analysis

The first stage of our module-oblivious domain management technique infers the activity of power domains during an application’s execution. Normally, a gate-level simulation could infer the activity of all processor gates for only one input set. However, our scalable hardware-software co-analysis presented in Chapter 4 uses a form of symbolic simulation that propagates Xs for all application inputs, allowing us to infer the activity of all gates for all possible input sets. Combined with the domain mapping that specifies which gates belong to each domain, we can infer domain activity for all possible executions of an application on a processor. Specifically, during the symbolic simulation, the simulator captures the activity of each gate at each point in the execution tree. A gate is considered active in a particular cycle if its value changes or if it has an unknown value (X) and is driven by an active gate; otherwise, the gate is idle. The resulting annotated symbolic execution tree describes all possible instances in which a gate could possibly toggle (and by extension, all instances in which each domain could possibly be active) for all possible executions of the application. As such, it also describes when
Figure 5.6: Our analysis generates input-independent power gating decisions for module-oblivious domains.
power domains (even module-oblivious domains) can be safely powered down and when they must be powered up. The next section describes how inferred domain activity information is translated into domain management decisions.

5.4.2 Gating Binary Annotation

Gating binary annotation (GBA) takes as input the annotated symbolic execution tree from gate activity analysis, gate-to-domain mapping information, and domain wakeup overheads, and produces a binary in which each static instruction is annotated with power gating decisions for all domains in the processor. Algorithm 3 describes GBA. GBA considers each path through the symbolic execution tree.\(^3\) During each cycle of a path’s execution, GBA determines which domains can have active gates and thus must be powered on. To ensure safety, GBA also marks a domain as active during the N cycles leading up to a period of activity, where N is the wakeup latency required to power up the domain. These cycle-level power gating decisions are mapped to all the static instructions that have dynamic instances in the pipeline during the wakeup cycles or the current cycle.

**Algorithm 3** Gating Binary Annotation for Power Gating Control

```plaintext
Procedure Annotate Binary with PG Decisions(annotated_symbolic_execution_tree, domain_mapping, domain_wakeup_overhead)
1. \(P_{SET} \leftarrow \) enumerate all paths in annotated_symbolic_execution_tree
2. Mark all domains as idle for all instructions/addresses in the binary
3. foreach path \(p \in P_{SET} \) do
4. foreach cycle \(c \in p \) do
5. foreach gate \(g \in \text{Processor} \) do
6. if \(g\) is toggled then
7. \(D \leftarrow \text{domain_mapping.get_domain}(g)\)
8. \(wo \leftarrow \text{domain_wakeup_overhead.get}(D)\)
9. \(I \leftarrow \text{get_instructions_being_executed}(p, c, wo)\)
10. foreach \(i \in I\) do
11. Mark domain \(D\) as active at instruction \(i\) in binary
12. end for
13. end if
14. end for
15. end for
16. end for
```

Once GBA has considered each execution path through an execution binary, each static instruction has an annotation specifying which domains must be powered on when the instruction is in the decode stage. This annotation guarantees safety, because each possible dynamic instance of a static

\(^3\)For our benchmarks, GBA takes 11.71 seconds, on average, and a maximum of 35.44 seconds for our largest benchmark.
Figure 5.7: Illustration of gating binary annotation for 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 wakeup latency of zero cycles.

instruction is considered by GBA. If a domain is marked as being powered on for any dynamic instance of a static instruction, the static instruction is annotated with an “ON” decision for the domain. This is conservative to ensure safety, but it works well for embedded applications, which tend to have simple control flow. If a domain is not active for any dynamic instance of a particular instruction (even considering wakeup overheads), the domain is powered off. The annotated binary containing domain management decisions can be used to manage power domains using one of the several techniques described in Section 5.4.4.

5.4.3 Illustrative Example

This section illustrates the procedures for managing module-oblivious domains, described in Sections 5.4.1 and 5.4.2, with an example. Figure 5.7 revisits the example code from Figure 5.3 to demonstrate that our technique based on hardware/software co-analysis can infer the activity of module-oblivious domains, which was impossible to infer from software alone.

Figure 5.7 shows the annotated symbolic execution tree generated by gate activity analysis (GAA). GAA simulates the application starting at instruc-
When an input value is read in instruction 3, instead of storing the input bits, unknown logic values (Xs) are stored in r15. During instruction 5, an X propagates to the PC inputs, since the result of the comparison in instruction 4 is unknown (X). At this point, a branch is created, and the simulation state is stored in a stack for later analysis with the address of instruction 8 (else:) in the PC inputs. Simulation continues through the left (then:) control flow path to completion, starting with instruction 6. After finishing instruction 9, the stored simulation state is popped off the stack and the right control flow path is simulated to completion, starting with instruction 8.

During simulation, GAA annotates each dynamic instruction with domain activity for each domain (D1 and D2 in Figure 5.7). ON means that at least one gate in the domain might be active during that instruction; OFF means that all of the domain’s gates are guaranteed to be inactive for that instruction. Next, Gating binary annotation (GBA) maps the domain states (ON/OFF states) from the symbolic execution tree to the static instructions in the application binary. Consider static instruction 1 (mov #0, r4). There is only one dynamic instance of the instruction in the symbolic execution tree, and for this instance, domain D1 is ON and D2 is OFF. Therefore, GBA annotates the corresponding static instruction with the information that D1 is ON and D2 is OFF.

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

Sections 5.4.1 and 5.4.2 describe a technique that can infer the activity of module-oblivious domains without costly hardware-based monitoring and use inferred domain activity to make safe and profitable domain management decisions. This section describes microarchitectural support for communicating domain management decisions to the control logic that powers the domains off and on.

**Power Gating Instructions:**
A straightforward way to generate power gating control signals is to insert instructions in the binary that direct power domains when to turn off and on. To ensure that a power domain is powered on before it is used, the wakeup instruction for a domain must arrive \( \text{wakeup-latency} \) cycles before an instruction, \( I_A \), that will activate the domain. For an in-order processor, we insert the wakeup instruction \( \text{wakeup-latency} \) 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 5.7 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 [118].

**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.

### 5.4.5 Ensuring Correctness

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

1. **Guaranteeing that domains turn on when needed:** Our co-analysis approach characterizes domain activity for all possible executions of an application for all possible inputs to the application. A power domain is only turned off if it is not used by an instruction in all execution paths through the code (Section 5.4.2).

2. **Guaranteeing that analysis is input-independent:** We perform a symbolic simulation in which all application inputs are replaced by $X$s. This ensures full characterization of application-induced activity on the processor for all possible application inputs (Section 5.4.1).

3. **Guaranteeing that hardware implementation is correct:** We use an automated industry-standard UPF flow to fully implement power gating designs and accurately account for all implementation overheads of power gating (Section 5.5.3).
Figure 5.8: Domain composition of module-based power domains. Each module belongs to only one domain.

5.5 Methodology

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

5.5.1 Constructing Power Domains

Module-Based Domains: We construct module-based domains following the conventional approach for aggressive power gating, in which power domains are formed to encompass microarchitectural modules. When the number of modules is greater than the number of allowable power domains, modules are grouped together into domains using hierarchical agglomerative clustering [119]. This clustering technique combines a set of N clusters into N-1 clusters, based on an optimization objective. In this case, the objective function uses activity profiles for the clusters (obtained from benchmark profiling) to determine which combination of modules maximizes the potential energy savings achieved by power gating the resulting domains. Potential energy savings are measured in gated cycles, where one gated cycle corresponds to power gating one gate in the gate-level netlist for one cycle. Figure 5.8 shows the domain composition for four module-based domains that maximize leakage energy savings for module-based power gating.
Module-Oblivious Domains: We use the same clustering technique as for module-based domains, with two key differences. First, whereas module-based domain construction begins with all processor *modules* in separate clusters and combines clusters using hierarchical agglomerative clustering to form the desired number of domains, module-oblivious domain construction begins with every *gate* in a separate cluster and combines clusters to form the desired number of domains. Since a gate may end up in a cluster containing gates from other modules, the resulting domains are module-oblivious.

Second, since an application’s in-field inputs may not always match the inputs used during profiling, we use activity profiles produced by input-independent gate activity analysis (Section 5.4.1) to identify correlated gates and generate power domains, instead of profiles captured assuming specific inputs. We treat an $X$ in an activity profile as a toggle, since it indicates that a net could toggle for some possible input. Input-independent domain formation ensures robustness of domains across variations in an application’s input set. We form domains using the activity profiles for only three randomly selected applications in our benchmark set (tea8, binSearch, Autocorr), and use these domains to perform evaluations for all thirteen benchmarks in Table 5.3. In practice, we envision that domains will be formed using activity profiles that are representative of a system’s target workloads (similar to how benchmarks are used to determine microarchitectural parameters). The actual workloads that the processor will run in the field may be different and many more than the number of benchmarks used for domain formation. As such, we chose a small number of benchmarks for domain formation relative to the total number of applications used for evaluation. Nevertheless, our evaluations show significant benefits even for the ten benchmarks that were not used for domain formation (Section 5.6). This is because the correlated activity among gates in different modules is often ISA and microarchitectural implementation-dependent, so only a small number of benchmarks are needed to determine which gates have correlated activity profiles and form domains accordingly.
5.5.2 Power Domain Management

Idle Count [120] is a hardware-based power gating management technique that uses a counter per domain to count the number of cycles a domain has been idle. The counter is reset every cycle its domain is active. When the counter reaches a threshold, \(k\), the domain is powered down. We perform evaluations for \(k = 5, 10,\) and \(100\). Although [120] only proposes power gating of functional units, we optimistically evaluate it for any arbitrary processor modules, even software-invisible modules. We only evaluate Idle Count for module-based domains, since activity monitoring is prohibitively expensive for module-oblivious domains (Section 5.3.2). To make this baseline even more optimistic, we do not consider the overhead of implementing hardware-based domain monitoring logic for software-invisible module-based domains.

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

Our approach, software-hardware co-analysis uses input-independent symbolic simulation to annotate instructions in the application binary with power gating decisions for each domain, as described in Section 5.4.

5.5.3 Simulation Infrastructure and Benchmarks

We verify our approach on a silicon-proven general-purpose processor – openMSP430 [63].\(^4\) Since MSP430 supports aggressive module-based power gating [94], it is a suitable testbed for comparison of module-based and module-oblivious power gating. The processor is synthesized, placed and routed in TSMC 65GP (65nm) technology at an operating point of 1V and 100 MHz using Synopsys Design Compiler [122] and Cadence EDI System [123]. Gate-level simulations are performed by running full benchmark applica-

\(^4\)MSP430 is one of the most popular processors used in low-power systems [94, 121].
Table 5.3: Benchmarks

<table>
<thead>
<tr>
<th>Embedded Sensor Benchmarks [64]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mult, binSearch, tea8, intFilt,</td>
<td></td>
</tr>
<tr>
<td>div, inSort, rle, tHold, intAVG</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EEMBC Embedded Benchmarks [65]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Autocorr, ConvEnc, FFT, Viterbi</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Overheads of implementing power gating (Isolation + Retention)

<table>
<thead>
<tr>
<th>Domain Type</th>
<th>Module-oblivious</th>
<th>Module-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Count</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Static Power (%)</td>
<td>1.95</td>
<td>2.48</td>
</tr>
<tr>
<td>Dynamic Power (%)</td>
<td>1.46</td>
<td>3.90</td>
</tr>
<tr>
<td>Area (%)</td>
<td>14.92</td>
<td>18.12</td>
</tr>
<tr>
<td>Wiring (%)</td>
<td>5.1</td>
<td>6.5</td>
</tr>
<tr>
<td>Delay (%)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Experiments were performed on a server housing two Intel Xeon E-2640 processors (eight cores each, 2GHz operating frequency, 64GB RAM).

5.5.4 Power Gating Implementation Overheads

Table 5.4 quantifies the overheads of implementing power gating with module-oblivious and module-based domains. Implementation overheads result from insertion of isolation and retention cells. Module-oblivious domains use more isolation cells than module-based domains and thus have higher overhead in terms of power (static and dynamic) and area. Also, module-oblivious domains increase wire length (e.g., for domain control logic), since cells in the
same domain may be spread out more across a chip. Despite having higher overhead, module-oblivious domains afford significantly higher power and energy reductions than module-based domains (Section 5.6). Furthermore, area and wiring overheads did not result in any change in cycle time, as the place and route tool was able to optimize the design for the same timing target. This is not surprising, considering that embedded processors are optimized for low power rather than high performance. Note that area and wiring overheads for module-oblivious domains can be reduced with domain-aware placement and routing optimizations that group cells that belong to the same domain [127]. Such optimizations are beyond the scope of current work.

5.6 Results

In this section, we evaluate and analyze the power benefits of module-oblivious power gating compared to aggressive module-based power gating. Note that results for module-based power gating are optimistic, since we allow even software-invisible modules to be power gated. We also do not account for any overhead for hardware-based monitoring logic for module-based power gating. Figure 5.9 compares the leakage energy savings provided by different power gating techniques. The stacked bars in the figure correspond to three different scenarios. The overall height of a stack shows the potential benefits of the technique when no implementation or instrumentation overheads are considered, i.e., the maximum potential benefits. The next level in a stacked bar shows benefits after static and dynamic overheads of domain isolation and state retention are accounted for, and the lowest level in a stack shows benefits when accounting for isolation, retention, and software instrumentation overheads. Note that we use the industry-standard UPF format to accurately account for the implementation overheads of power gating. Note also that for our instrumentation overheads, we have conservatively assumed the software approach with the highest overhead – binary instrumentation with dedicated power gating instructions (Section 5.4.4) – and have accounted for both static and dynamic energy overheads.

**Module-Oblivious vs. Module-Based Domains:**

Figure 5.9 shows that power gating module-oblivious domains can provide
Figure 5.9: Comparison of leakage energy savings provided by different domain management and formation techniques for different numbers of power domains. Results in each stack (from top to bottom) correspond to maximum potential benefits of the technique, benefits after accounting for isolation and retention overheads, and benefits after accounting for isolation, retention, and instrumentation overheads.
Figure 5.10: These “cool” maps compare the potential for power gating between module-based and module-oblivious domains. Cooler colors represent that the logic in a domain is idle together and has more potential to be power gated.
Figure 5.11: Most of the energy savings provided by module-oblivious domains are contributed by only a small number of mostly-idle domains.
significant benefits over conventional module-based domains. On average, power gating on module-oblivious domains provides $1.4 \times$ more leakage savings than the maximum savings that can be achieved with module-based domains (Oracle (Module-based)) and $2 \times$ more savings than hardware-based management of module-based domains.\(^5\) Figure 5.10 provides a visualization that explains why module-oblivious domains provide more opportunities for power gating than module-based domains. The figure is a type of correlation matrix that shows the power gating correlation between different sub-module pairs (sub-module\(_1\), sub-module\(_2\)) in the processor,\(^6\) where power gating correlation is defined as the fraction of cycles that the two sub-modules, sub-module\(_1\) and sub-module\(_2\), are both idle at the same time. We have defined the color scale such that cooler colors mean that the sub-modules are more frequently idle at the same time and therefore can be power gated together.

Figure 5.10a shows the power gating correlation for module-based domains, and Figure 5.10b is for module-oblivious domains. The different sub-modules in the two matrices are arranged such that sub-modules belonging to the same domain form adjacent rows and columns. The dashed boxes along the main diagonal encircle all the power gating correlation scores for pairs of sub-modules that belong to the same power domain (Figure 5.4 shows the composition of each module-oblivious domain, and Figure 5.8 shows the composition of each module-based domain).

Module-based domains do not account for the fact that different parts of the same microarchitectural modules may have uncorrelated activity profiles; as a result, they provide fewer opportunities for power gating. A single sub-module (even a single gate!) with high activity or uncorrelated idle times can sabotage power gating opportunities for an entire domain. For example, even though large portions of the domains in Figure 5.10a are “cool”, the small number of “hot” cells in each domain prevent many power gating opportunities for the domains. Figure 5.10a shows that in many cases, moving a small number of gates to a different domain could provide more opportunities for power gating larger areas of logic for longer periods of time. This explains the significant improvement in benefits seen in Figure 5.9 for

\(^5\)Note that these results correspond to full-fledged UPF implementations of power gating that account for all implementation overheads.

\(^6\)Each sub-module corresponds to one of the module partitions represented as pie sections in Figure 5.4.
module-oblivious power gating over module-based power gating. By forming domains that contain logic from different modules with similar activity profiles, module-oblivious domains do not allow more active logic to ruin power gating opportunities for less active logic in the same module.

Managing Module-Oblivious Domains:

While module-oblivious domains provide significant potential for power benefits, they cannot be managed by conventional software- or hardware-based management techniques (Section 5.3). Figure 5.9 compares the benefits of the proposed software-hardware co-analysis technique for managing module-oblivious domains, which we refer to hereafter as co-analysis, against oracular management. Oracular management assumes perfect knowledge of inputs to make optimal management decisions that exploit every possible cycle of profitable power gating. Co-analysis, on the other hand, uses Xs for inputs to guarantee that power gating decisions will be safe for all possible inputs, since actual inputs are not known at compile time, when co-analysis is performed. Also, since inputs can affect the control paths taken through a program, co-analysis only decides to power gate at a specific point (static instruction) in a program when a domain will not be activated by any possible control path flowing through that point. This ensures safety under all scenarios, even input-dependent data and control. In spite of this conservative approach, results show that co-analysis is a very effective management technique for module-oblivious domains, as its power benefits are within 8% of optimal (oracle) management of module-oblivious domains.

Co-analysis can be used to generate domain management decisions for any set of power domains, no matter how they are formed. We evaluate co-analysis also for module-based domains and compare the benefits achieved against those achieved by state-of-the-art hardware-based management (Idle Count). Figure 5.9 shows that co-analysis can save (12%) more energy than hardware-based domain management for module-based domains, even though we assume no hardware overhead for implementing Idle Count. In fact, the benefits of co-analysis are within 6% of optimal (oracle) for module-based domains. Co-analysis has an advantage over hardware-based domain management even for module-based domains, since co-analysis uses application information to create a tailored power gating strategy for each application, whereas a hardware-based technique necessarily uses the same hard-wired power gating strategy for all applications. Since hardware-based
management techniques must apply the same strategy to different applications or application phases that may have different patterns of activity and idleness, they may miss opportunities when power gating is applied too conservatively or incur overheads when power gating is applied too aggressively. For example, a domain managed by Idle Count necessarily spends a fraction of a profitable idle period powered up, as it counts idle cycles before deciding to power down. Also, if a power domain goes to sleep but is needed by an application in fewer cycles than its wakeup latency, the energy penalty for wakeup can cause negative energy savings. In short, the hardware controller for Idle Count must guess the length of idle periods without knowing whether they will be longer than the break-even point. Co-analysis, on the other hand, makes application-aware annotation decisions that account for the break-even point, so it can apply power gating aggressively without ever causing negative energy savings. Finally, note that our implementation is for 65nm technology (Section 5.5), where the leakage power is only 29% of total power. The benefits of our technique are expected to increase for lower (planar) technology nodes, where the problem of leakage power increases significantly [82, 83].

**Sensitivity Analysis:**
We also compare the power gating techniques across different numbers of power domains and different domain wakeup latencies. The sub-figures of Figure 5.9 compare the leakage reduction benefits of different power gating techniques for different numbers of domains. Since more domains imply increased specialization of each domain and better adaptation to the activity profile of an application, the potential benefits of power gating generally increase as the number of domains increases. However, our analysis shows that benefits vary by only a few percent for two, three, or four domains. The reason for this behavior is that most of the benefits provided by power gating come from powering down logic with long, correlated idle periods. As illustrated in Figure 5.10, module-oblivious domains enhance such power gating opportunities by relocating logic that would sabotage power gating opportunities for a domain to a different domain with more correlated behavior. This has the effect of collecting logic from different modules into one or more domains that are almost always off. Since only two domains are needed to divide logic into mostly off and mostly on domains, only two module-oblivious domains are needed to achieve most of the benefits that they can provide.
Table 5.5: Performance overhead (%) introduced by different power domain management techniques.

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

Figure 5.11 illustrates this point by showing the percentage of energy savings that each module-oblivious domain contributes to the total. For two, three, and four domains, only one of the domains contributes the majority (96%) of savings (the coolest domain in Figure 5.10b). This motivates a design with only two domains, since domain isolation and management costs are lower for fewer domains. Module-based domains show similar behavior; however, the presence of logic with uncorrelated activity within some modules limits the length of idle periods and consequently the benefits that can be achieved with module-based domains.

The different clusters of bars within each sub-figure of Figure 5.9 compare the leakage reduction benefits of different power gating approaches for different wakeup latencies (1, 3, and 10 cycles). As a result, potential benefits decrease with longer wakeup latencies.

Shorter wakeup latencies allow power gating to be applied aggressively for shorter idle periods, but this may increase instrumentation overhead due to frequent powering down and up of domains. Our application-aware co-analysis approach accounts for instrumentation overhead and wakeup latency during binary annotation to ensure that a power domain is only powered down when the net effect reduces energy. On average, the time between con-

---

7 The domain wakeup latency of 1 cycle is the most realistic for our small embedded processor [128, 101]. We also evaluated 100- and 1000-cycle wakeup latencies in our sensitivity analysis; however, we omitted the results since they showed the same trend as 10-cycle results. Since, it is only profitable to power down a domain if it will be idle for longer than the wakeup latency, wakeup latency has the effect of a low-pass filter on domain power-down/up decisions during analysis. I.e., a given wakeup latency filters out idle periods that are shorter than the wakeup latency.
secutive power gating decisions is 98 cycles, while the minimum and maximum times between decisions are 5 and 8127 cycles, respectively, demonstrating that correlated activity across gates in different modules often exists at a coarse time granularity. Table 5.5 characterizes the performance impact of instrumentation overhead for different numbers of domains and different wakeup latencies, and compares against the performance overheads introduced by hardware-based domain management (Idle Count). In the table, we present the overhead of our most costly binary instrumentation technique (inserting power gating instructions) under the column titled “Co-analysis”. Since inserting power gating instructions increases runtime, the column represents both performance and energy overhead. Results show that co-analysis has lower performance overhead than Idle Count for low idle count thresholds. While Idle Count has slightly lower performance overhead than co-analysis for a large idle count threshold (100), it loses 100 cycles of potential power gating opportunity for each idle period. In any case, the leakage savings of co-analysis significantly outweigh those of Idle Count (Figure 5.9), since the hardware-based technique cannot be used to manage module-oblivious domains without incurring prohibitively large implementation overheads.

5.7 Summary

In this chapter, we showed that module-oblivious power gating can provide significantly more leakage savings than state-of-the-art aggressive module-based power gating by allowing larger areas of a processor to be powered down for longer periods of time. Since conventional software- and hardware-based management techniques cannot be applied for module-oblivious power gating, we presented a novel technique for low-cost management of module-oblivious power domains in embedded processors, based on input-independent hardware-software co-analysis. Our technique is automated, does not require programmer intervention, and incurs low management overhead. We demonstrated that module-oblivious power gating based on our technique reduces leakage energy by $2 \times$ with respect to state-of-the-art aggressive module-based power gating for a common embedded processor. Our technique for management of module-oblivious power domains achieves leakage energy savings that are within 8% of those achieved by optimal oracular management. Finally, our
technique for managing module-oblivious domains is effective even at managing conventional module-based domains. It saves 12% more energy than an idealized implementation of Idle Count – a hardware-based domain management technique for module-based domains. Our benefits are within 6% of optimal for module-based domains.
IoT applications not only require low power, but also require low cost—easy development of new applications and low area to fit on new substrates. For design and development cost reasons, these applications rely on ultra-low-power general-purpose microcontrollers and microprocessors, making such processors the most abundant type produced and used today. While general-purpose processors have several advantages, such as amortized development cost across many applications, they are significantly over-provisioned for many area- and power-constrained systems, which tend to run only one or a small number of applications over their lifetime. In this chapter, we make a case for bespoke processor design, an automated approach that tailors a general-purpose processor IP to a target application by removing all gates from the design that can never be used by the application. Since removed gates are never used by an application, bespoke processors can achieve significantly lower area and power than their general-purpose counterparts without any performance degradation. Also, gate removal can expose additional timing slack that can be exploited to increase area and power savings or performance of a bespoke design. Bespoke processor design reduces area and power by 62% and 50%, on average, while exploiting exposed timing slack improves average power savings to 65%.

6.1 Introduction

A large class of emerging applications is characterized by severe area and power constraints. For example, wearables [66, 70] and implantables [129, 130] are extremely area- and power-constrained. Several IoT applications, such as stick-on electronic labels [131], RFIDs [132], and sensors [67, 81], are also extremely area- and power-constrained. Area constraints are expected
to be severe also for printed plastic [133] and organic [134] 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 [2, 3, 4], in spite of their well-known inefficiency compared to ASIC and FPGA-based solutions [135]. 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 (Section 6.2). Also, these unused gates continue to consume power, resulting in significant power inefficiency. While adaptive power management techniques (e.g., power gating [136, 102]) 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 (Section 6.6). These techniques also worsen area inefficiency.

One approach to significantly increase the area and power efficiency of a microprocessor for a given application is to eliminate 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 tailored to the application – a bespoke processor – 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 re-
Figure 6.1: General purpose processors are overdesigned for a specific application (top). A bespoke processor design methodology allows a microprocessor IP licensor or licensee to target different applications efficiently without additional software or hardware development cost (bottom).

quire programmer intervention or hardware support either, since the software application can still run, unmodified, on the bespoke processor.

In this chapter, we present a methodology to automatically generate a bespoke processor for an application out of a general-purpose processor / microcontroller IP core. Our methodology relies on gate-level symbolic simulation to identify gates in the microprocessor IP that cannot be toggled by the application, irrespective of the application inputs, and automatically eliminates them from the design to produce a significantly smaller and lower power design with the same performance. In many cases, reduction in the number of gates also introduces timing slack that can be exploited to improve performance or further reduce power and area. Since the original design is pruned at the granularity of gates, the resulting methodology is much more effective than any approach that relies on coarse-grained application-specific customization. The proposed methodology can be used either by IP licensors or IP licensees to produce bespoke designs for the application of interest (Figure 6.1). Simple extensions to our methodology can be used to generate bespoke processors that can support multiple applications or different degrees of in-field software programmability, debuggability, and updates.
This chapter makes the following contributions:

- We propose **bespoke processors** – a novel approach to reducing area and power by tailoring a processor to an application, such that a processor consists of only those gates that the application needs for any possible execution with any possible inputs. A bespoke processor still runs the unmodified application binary without any performance degradation.
- We present an automated methodology for generating bespoke processors. Our symbolic gate-level simulation-based methodology takes the original microprocessor IP and application binary as input to produce a design that is functionally equivalent to the original processor from the perspective of the target application while consisting of the minimum number of gates needed for execution.
- We quantify the area and power benefits of bespoke processors for a suite of sensor and embedded benchmarks. Area reductions are up to 92% (46% minimum, 62% on average) and power reductions are up to 74% (37% minimum, 50% on average) compared to a general-purpose microprocessor. When timing slack resulting from gate removal is exploited, power reductions increase to up to 91% (50% minimum, 65% on average).
- Finally, we present and analyze design approaches that can be used to support bespoke processors throughout the product life-cycle. These design approaches include 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.

### 6.2 Motivation

Area- and power-constrained microprocessors and microcontrollers are the most abundant type of processor produced and used today, with projected deployment growing exponentially in the near future [2, 3, 4, 137]. This explosive growth is fueled by emerging area- and power-constrained applications, such as the internet-of-things (IoT), wearables, implantables, and sensor networks. The microprocessors and microcontrollers used in these applications are designed to include a wide variety of functionalities in order to support a large number of diverse applications with different requirements.
Figure 6.2: 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.

On the other hand, the embedded systems designed for these applications typically consist of one application or a small number of applications, running over and over on a general-purpose processor for the lifetime of the system [138]. Given that a particular application may only use a small subset of the functionalities provided by a general-purpose processor, there may be a considerable amount of logic in a general-purpose processor that is not used by an application. Figure 6.2 illustrates this point, showing the fraction of gates in an openMSP430 [63] processor that are not toggled when a variety of applications (Table 6.1) are executed on the processor with many different input sets. The bar in the figure shows the intersection of all gates that were not exercised (toggled) by the application for any input, and the interval shows the range in fraction of unexercised gates across different inputs. For each application, a significant fraction (around 30% - 60%) of the processor’s gates were not toggled during any execution of the application. These results indicate that there may be an opportunity to reduce area and power significantly in area- and power-constrained systems by removing logic from the processor that cannot be exercised by the application(s) running on the processor, if it can be guaranteed that removed logic will never be needed for any possible execution of the application(s).

However, identifying all the logic that is guaranteed to never be used by an application is not straightforward. One possible approach is profiling,
wherein an application is executed for many inputs and the set of gates that were never exercised is recorded, as in Figure 6.2. However, profiling cannot guarantee that the set of gates used by an application will not be different for a different input set. Indeed, profiling results in Figure 6.2 show considerable variations in exercised gates (up to 13%) for different executions of the same application with different inputs. Thus, an application might require different gates and execute incorrectly for an unprofiled input.

Static application analysis represents another approach for determining unusable logic for an application. However, application analysis may not identify the maximum amount of logic that can be removed, since unused logic does not correspond only to software-visible architectural functionalities (e.g., arithmetic units), but also to fine-grained and software-invisible microarchitectural functionalities (e.g., pipeline registers). For example, consider two different applications – FFT and binSearch. Figure 6.3 shows the gates in the processor that were not exercised during any profiling execution of the applications. Since the applications use different subsets of the functionalities provided by the processor, the parts of the processor that they do not exercise are different. However, a closer look reveals that while some of the differences correspond to coarse-grained software-visible functionalities (e.g., the multiplier is used by FFT but not by binSearch), other differences are fine-grained, software-invisible, and cannot be determined through application analysis (e.g., different gate-level activity profiles in modules like the processor frontend). As another example, Figure 6.4 shows the breakdown of instructions used by intFilt and scrambled-intFilt. The two applications use exactly the same instructions (scrambled-intFilt is a synthetic benchmark generated by scrambling instructions in intFilt); however, the die graphs in Figure 6.4 show that the sets of unexercised gates for the applications are different. This is due to the fact that even the sequence of instructions executed by an application can influence which logic the application can exercise in a processor depending on the microarchitectural details. Such interactions cannot be determined simply through application analysis.

Given that the fraction of logic in a processor that is not used by a given application can be substantial, and many area- and power-constrained systems only execute one or few applications for their entire lifetime, it may be possible to significantly reduce area and power in such systems by removing logic from the processor that cannot be used by the application(s). How-
Figure 6.3: Gates not toggled by two applications – (a) FFT and (b) binSearch – for profiling inputs. Gray gates are not toggled by either application. Red gates are unique untoggled gates for each application.

(a) FFT    (b) binSearch

Figure 6.4: Gates not toggled by (a) intFilt and (b) scrambled intFilt for the same input set. Gray gates are not toggled by either application. Red gates are unique untoggled gates for each application. Even though the applications use the same set of instructions and control flow, the gates that they exercise are different.

(a) intFilt    (b) Scrambled intFilt
ever, since different applications can exercise substantially different parts of a processor, and simply profiling or statically analyzing an application cannot guarantee which parts of the processor can and cannot be used by an application, tailoring a processor to an application requires a technique that can identify all the logic in a processor that is guaranteed to never be used by the application and remove unusable logic in a way that leaves the functionality of the processor unchanged for the application. In the next section, we describe a methodology that meets these requirements. We call general-purpose processors that have been tailored to an individual application bespoke processors, reminiscent of bespoke clothing, in which a generic clothing item is tailored for an individual person.

6.3 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, 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 6.5 shows our process for tailoring a bespoke processor to a target application. The first step – input-independent gate activity analysis – performs a type of symbolic simulation, where unknown input values are represented as Xs, and gate-level activity of the processor is characterized for all possible executions of the application, for any possible inputs to the application. 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
Figure 6.5: 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.

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.

6.3.1 Input-Independent Gate Activity 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 the symbolic simulation-based co-analysis technique described in Chapter 4. This analysis 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 ($X$s), which are treated as both 1s and 0s when recording possible toggled gates. 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. The result of input-independent gate activity analysis for an application is a list of all gates that cannot be toggled in any execution of the application, along
with their constant values. Since the logic functions performed by these gates are not necessary for the correct execution of the binary for any input, they may safely be cut from the netlist, as long as their constant output values are preserved. The following section describes how unusable gates can be cut from the processor without affecting the functionality of the processor for the target application.

6.3.2 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 bespoke processor. Figure 6.6 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.
6.3.3 Illustrative Example

This section illustrates how bespoke processor design tailors a processor design to a particular application, as described in Sections 6.3.1 and 6.3.2. Figure 6.7 illustrates the bespoke design process. The left part of Figure 6.7 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.

6.3.4 Correctness

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 \( B_A \) of processor \( P \) tailored to an application \( A \) is functionally equivalent to processor \( P \) with respect to application \( A \); \( B_A \) produces the same output as \( P \) for any possible execution of \( A \).

**Proof**: The first step in creating \( B_A \) – input-independent gate activity analysis (Section 6.3.1) – identifies the subset \( E \) of all gates in the processor that can possibly be exercised by \( A \), for all possible inputs. The analysis also identifies the constant output values for all gates \( U \) that can never be exercised by \( A \). It follows that \( E \cap U = \emptyset \) and \( E \cup U = G \), where \( G \) is the set of all gates in \( P \). Cutting and stitching (Section 6.3.2) removes all gates in the set \( U \) and ties their output nets to their known constant values, such that the functionality of all gates in \( U \) is maintained in \( B_A \). All gates in \( E \) remain in the bespoke design, so all gates in \( E \) have the same functionality and produce the same outputs in \( B_A \) and \( P \). Since \( E \cup U = G \), it follows that \( B_A \) is functionally equivalent to \( P \) for \( A \) and produces the same output as \( P \) for all possible inputs to \( A \). ■

We also verified correctness through input-independent gate activity anal-
ysis and input-based simulations on both the original and bespoke processor for every application (Section 6.5.1), confirming that outputs of both processors were the same in each case.

6.3.5 Supporting Multiple Applications

While bespoke processor design involves tailoring a general-purpose processor into an application-specific processor implementation, bespoke processors, which are descended from general-purpose processors, still retain some programmability. In this section, we describe several approaches for creating bespoke processors that support multiple applications.

The first and most straightforward case is when the multiple target applications are known a priori at design time. For example, a licensor or licensee (see Figure 6.1) that wants to amortize the expense of designing and manufacturing a chip may choose to tailor a bespoke processor to support multiple applications. In this case, the bespoke processor design methodology (Figure 6.5) is simply expanded to support multiple target applications, as shown in Figure 6.8. Given a known set of binaries that need to be supported, gate activity analysis is performed for each application, and cutting and stitching is performed for the intersection of unused gates for the applications. The intersection of unused gates represents all the gates that are not used by any of the target applications. The resulting bespoke processor contains all the gates necessary to run any of the target applications. While there may be some area and power cost compared to a bespoke design for a single application due to having more gates, Section 6.5 shows that a bespoke processor supporting multiple applications still affords significant area and power benefits.

There may also be cases where it is desirable for a bespoke processor to support an application that is not known at design time. For example, one advantage of using a programmable processor is the ability to update the target application in the field to roll out a new software version or to fix a software bug. Tailoring a bespoke processor to a specific application invari-

---

\textsuperscript{1}Industrial equivalence checking tools (e.g., Formality [139]) check static equivalence (i.e., their analysis is application-independent); however, the bespoke and original designs are only equivalent for the target application, not in general. Therefore, we rely on gate-level analysis and input-based simulations for verification.
Figure 6.8: To support multiple programs, our bespoke design technique performs input-independent gate activity analysis on each program. Cutting and stitching is performed using the intersection of the untoggled gates lists from all supported programs.

ably reduces its ability to support in-field updates. However, even in the case when an application was unknown at design time, it may be possible for a bespoke processor to support the application. For instance, it is always possible to check whether a new software version can be supported by a bespoke processor by checking whether the gates required by the new software version are a subset of the gates in the bespoke processor. It may be possible to increase coverage for in-field updates by anticipating them and explicitly designing the processor to support them. As an example, this may be used to support common bug fixes by automatically creating mutants of the original software by injecting common bugs [140], then creating a bespoke design that supports all the mutant versions of the program. This approach may increase the probability that a debugged version of the software is supported by the bespoke design.

Sometimes an in-field update represents a significant change that is beyond the scope of a simple code mutation. To support such updates, a bespoke processor may need to provide support for a small number of ISA features that can be used to implement arbitrary computations - e.g., a Turing-complete instruction (or set of instructions), in addition to the target application(s). Consider adding support for \texttt{subneg}, an example Turing-complete instruc-
Figure 6.9: Functionality and MSP430 implementation of Turing-complete \texttt{subneg} pseudo-instruction.

The memory operand addresses and the branch target are assumed to be unknown values (Xs), a binary that characterizes the behavior of \texttt{subneg} can be co-analyzed with the target application binary to tailor a Turing-complete bespoke processor that supports the target application natively and can handle arbitrary updates, possibly with some area, power, and performance overhead.

6.4 Methodology

6.4.1 Simulation Infrastructure and Benchmarks

We verify our technique on a silicon-proven processor – openMSP430 [63], an open-source version of one of the most popular ULP processors [121, 94]. The processor is synthesized, placed, and routed in TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [122] and Cadence EDI System [123]. Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using a custom gate-level simulator that efficiently traverses the control flow graph of an application and captures input-independent activity profiles (Section 6.3.1). Table 6.1 lists our benchmark applications. We show results for all benchmarks from [64] and all EEMBC benchmarks [65]

136
that fit in the program memory of the processor. We also added unit test benchmarks [63] corresponding to some functionalities in the processor that were not exercised by the other benchmarks. In addition to evaluating a bare-metal environment common in area- and power-constrained embedded systems [73, 74, 75, 76], we also evaluate bespoke processors that support our applications running on the processor with an operating system (FreeRTOS [77]). Benchmarks are chosen to be representative of emerging ultra-low-power application domains such as wearables, internet of things, and sensor networks [64]. Also, benchmarks were selected to represent a range of complexity in terms of control flow and execution length. Power analysis is performed using Synopsys Primetime [126]. Experiments were performed on a server housing two Intel Xeon E-2640 processors (eight cores each, 2GHz operating frequency, 64GB RAM).

### 6.4.2 Baselines

For evaluations, we compare bespoke designs against two baseline processors. The first baseline is the openMSP430 processor, optimized to minimize area and power for operation at 100 MHz and 1V. For the second baseline, we create aggressively optimized application-specific versions of openMSP430 for

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Max Execution Length (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>Binary search</td>
<td>2037</td>
</tr>
<tr>
<td>div</td>
<td>Unsigned integer division</td>
<td>402</td>
</tr>
<tr>
<td>inSort</td>
<td>In-place insertion sort</td>
<td>4781</td>
</tr>
<tr>
<td>intAVG</td>
<td>Signed integer average</td>
<td>14512</td>
</tr>
<tr>
<td>intFilt</td>
<td>4-tap signed FIR filter</td>
<td>113791</td>
</tr>
<tr>
<td>mult</td>
<td>Unsigned multiplication</td>
<td>210</td>
</tr>
<tr>
<td>rle</td>
<td>Run-length encoder</td>
<td>8283</td>
</tr>
<tr>
<td>tHold</td>
<td>Digital threshold detector</td>
<td>18511</td>
</tr>
<tr>
<td>tea8</td>
<td>TEA encryption algorithm</td>
<td>2228</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier transform</td>
<td>406006</td>
</tr>
<tr>
<td>Viterbi</td>
<td>Viterbi decoder</td>
<td>1167298</td>
</tr>
<tr>
<td>convEn</td>
<td>Convolutional encoder</td>
<td>117789</td>
</tr>
<tr>
<td>autocorr</td>
<td>Autocorrelation</td>
<td>8092</td>
</tr>
<tr>
<td>irq</td>
<td>Interrupt test</td>
<td>210</td>
</tr>
<tr>
<td>dbg</td>
<td>Debug interface</td>
<td>14166</td>
</tr>
</tbody>
</table>
Figure 6.10: 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.

Each benchmark application can be rewritten to remove unused modules before performing synthesis, placement, and routing. Such an approach is representative of an Xtensa-like approach [142], where the processor configuration is customized for a particular application. Note, however, that our baseline is significantly more aggressive than an Xtensa-like approach, since it requires our input-independent gate activity analysis technique (Section 6.3.1) to identify the modules that cannot be used by an application. Such module identification may not be possible through static analysis of application code alone.

6.5 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 6.10 shows the fraction of gates in the original processor design that could be toggled by each benchmark. 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

---

2 Unlike Figure 6.2, which presents results from profiling, Figure 6.10 shows results from input-independent gate analysis.
the baseline design. At most, 57% of the gates in the baseline design can 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.

Bespoke processors have fewer gates, lower area, and lower power than their general-purpose counterparts. Figure 6.11 shows the reduction in gates, area, and power afforded by bespoke processors tailored to each benchmark.

Figure 6.11: Reduction (%) in gate count, area, and power for a bespoke design, compared to the baseline processor.
Figure 6.12: Reduction (%) in gate count, area, and power for bespoke designs, compared to application-specific coarse-grained module-level bespoke design.

**FFT**, which has the smallest gate count reduction (44%), still reduces area by 47% and power by 37%, relative to the baseline design. Area savings are up to 92% (\texttt{dbg}), while power savings are up to 74% (\texttt{dbg}).

Figure 6.10 shows that some modules could be wholly removed for specific benchmarks (e.g., the multiplier can be removed for \texttt{binSearch}, since it cannot use any gates in the multiplier). For such modules, it is possible to use an Xtensa-like approach [142], 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 6.12 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 removes gates from critical paths, exposing additional timing slack that can be exploited for energy savings. Table 6.2 shows timing slack

---

3Note that gate count reduction reported in Figure 6.11 is different than fraction of toggled gates in Figure 6.10, since bespoke design also removes some toggled gates that cannot propagate their toggles to state elements or output ports.
Table 6.2: Benefits of exploiting timing slack created by cutting, stitching, and re-synthesis.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Timing Slack (%)</th>
<th>$V_{min}$ (V)</th>
<th>Addl. Power Savings from Slack (%)</th>
<th>Total Power Savings (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>24.30</td>
<td>0.81</td>
<td>36.86</td>
<td>72.1</td>
</tr>
<tr>
<td>div</td>
<td>24.51</td>
<td>0.82</td>
<td>34.53</td>
<td>69.9</td>
</tr>
<tr>
<td>inSort</td>
<td>22.24</td>
<td>0.83</td>
<td>33.25</td>
<td>67.6</td>
</tr>
<tr>
<td>intAVG</td>
<td>23.37</td>
<td>0.83</td>
<td>33.35</td>
<td>67.7</td>
</tr>
<tr>
<td>intFilt</td>
<td>23.23</td>
<td>0.84</td>
<td>31.31</td>
<td>58.5</td>
</tr>
<tr>
<td>mult</td>
<td>20.45</td>
<td>0.91</td>
<td>20.33</td>
<td>59.3</td>
</tr>
<tr>
<td>rle</td>
<td>22.10</td>
<td>0.83</td>
<td>33.31</td>
<td>66.8</td>
</tr>
<tr>
<td>tHold</td>
<td>24.07</td>
<td>0.81</td>
<td>36.90</td>
<td>73.5</td>
</tr>
<tr>
<td>tea8</td>
<td>23.55</td>
<td>0.83</td>
<td>33.23</td>
<td>65.7</td>
</tr>
<tr>
<td>FFT</td>
<td>21.74</td>
<td>0.90</td>
<td>20.13</td>
<td>50.0</td>
</tr>
<tr>
<td>Viterbi</td>
<td>23.48</td>
<td>0.83</td>
<td>33.18</td>
<td>64.9</td>
</tr>
<tr>
<td>convEn</td>
<td>23.96</td>
<td>0.83</td>
<td>33.03</td>
<td>63.9</td>
</tr>
<tr>
<td>autocorr1</td>
<td>18.48</td>
<td>0.91</td>
<td>18.31</td>
<td>50.3</td>
</tr>
<tr>
<td>irq</td>
<td>17.91</td>
<td>0.92</td>
<td>16.24</td>
<td>57.7</td>
</tr>
<tr>
<td>dbg</td>
<td>45.70</td>
<td>0.60</td>
<td>67.73</td>
<td>91.5</td>
</tr>
</tbody>
</table>
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.\(^4\) Table 6.2 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.

### 6.5.1 Verification

We followed a two-pronged approach to verify our bespoke processor designs. First, we performed input-independent gate activity analysis on the bespoke processor design and compared the processor state between the original and bespoke processors in each cycle. At the end of activity analysis, we compared the contents of the data memory with that of the original design to ensure that both designs produced the same outputs. There were no discrepancies for any of the bespoke processors, indicating that the bespoke processors traverse the same states as the original processor and produce the same outputs for each benchmark. While this verification efficiently\(^5\) checks for functional correctness considering all possible inputs, it does not explicitly guarantee that the data memory contents of a bespoke processor are correct for any specific inputs. For an explicit proof of the correctness of bespoke processors, see Section 6.3.4.

The second method we used to verify bespoke processors involves performing input-based simulations on the original and bespoke processors to confirm that they produce the same outputs. Outputs produced during simulation are recorded, and the outputs and data memory from the original and bespoke processors are compared for equivalence. Since it is infeasible to simulate the application with all possible inputs, we used KLEE LLVM Execution Engine (KLEE) [143] to generate inputs that exercise as many control paths through the application as possible. Table 6.3 lists the number of inputs simulated for each benchmark and the corresponding coverage of

\(^4\)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.

\(^5\)Table 6.3 shows that the runtime of X-based simulations is within an order of magnitude of a single input-based simulation.
Table 6.3: Verification runtime and coverage.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Sim. Time (s)</th>
<th>X-Based Input</th>
<th>Num Paths</th>
<th>Line %</th>
<th>Br. %</th>
<th>Br. Dir. %</th>
<th>Gate %</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>23</td>
<td>3</td>
<td>83</td>
<td>100</td>
<td>100</td>
<td>93</td>
<td>87</td>
</tr>
<tr>
<td>div</td>
<td>7</td>
<td>22</td>
<td>1</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>93</td>
</tr>
<tr>
<td>inSort</td>
<td>25</td>
<td>156</td>
<td>718</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>93</td>
</tr>
<tr>
<td>intAVG</td>
<td>116</td>
<td>79</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>82</td>
</tr>
<tr>
<td>intFilt</td>
<td>1365</td>
<td>625</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>28</td>
</tr>
<tr>
<td>mult</td>
<td>20</td>
<td>20</td>
<td>1</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>rle</td>
<td>20</td>
<td>32</td>
<td>1</td>
<td>74</td>
<td>100</td>
<td>75</td>
<td>92</td>
</tr>
<tr>
<td>tHold</td>
<td>7</td>
<td>27</td>
<td>6239</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>88</td>
</tr>
<tr>
<td>tea8</td>
<td>21</td>
<td>32</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>93</td>
</tr>
<tr>
<td>FFT</td>
<td>10498</td>
<td>5669</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>47</td>
</tr>
<tr>
<td>Viterbi</td>
<td>433</td>
<td>3637</td>
<td>8</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>86</td>
</tr>
<tr>
<td>convEn</td>
<td>1401</td>
<td>168</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>89</td>
</tr>
<tr>
<td>autocorr</td>
<td>90</td>
<td>13</td>
<td>1</td>
<td>38</td>
<td>14</td>
<td>14</td>
<td>71</td>
</tr>
</tbody>
</table>

the code. For most benchmarks, all lines and branch directions are covered. Where coverage is not 100%, the portion of the code that was not covered was not executable. The table also reports the fraction of gates in the bespoke designs that were exercised during the input-based simulations. We see that a majority of the gates (78%, on average) were toggled during the simulations, indicating that the majority of gates in a bespoke design are necessary.\footnote{Note that gate coverage is not expected to be 100%, since KLEE aims to cover lines of code and execution paths, not gates. In particular, benchmarks that use the multiplier (intFilt, mult, FFT, and autocorr) see low gate coverage since the multiplier is a significant fraction of bespoke designs for such benchmarks and KLEE does not try to form inputs to the multiplier to increase datapath coverage.} Table 6.3 also shows the aggregate runtime of the input-based simulations, providing some quantification of verification effort.\footnote{Note that simulations for multiple inputs can easily be parallelized to reduce the verification time significantly.}

### 6.5.2 Supporting Multiple Programs

Bespoke processors are able to support multiple programs by including the union of gates needed to support all of the programs. Figure 6.13 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 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...
Figure 6.13: Normalized gate count, area, and power ranges for all possible bespoke processor supporting multiple applications.

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 \texttt{dbg} and \texttt{irq}. Each application uses components of the processor that are not exercised by the other program; \texttt{dbg} exercises the debug module, while \texttt{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.

### 6.5.3 Supporting In-Field Updates

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.

In-field updates may often be deployed to fix minor correctness bugs (e.g., off-by-one errors, etc.) [144]. To emulate in-field updates to fix bugs, we use
Table 6.4: MILU produces three types of mutants. Type I: Logical conditional operator mutants. Type II: Computation operator mutants. Type III: Loop conditional operator mutants.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type I</th>
<th>Type II</th>
<th>Type III</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>inSort</td>
<td>8</td>
<td>0</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>rle</td>
<td>0</td>
<td>20</td>
<td>25</td>
<td>45</td>
</tr>
<tr>
<td>tea8</td>
<td>48</td>
<td>24</td>
<td>10</td>
<td>82</td>
</tr>
<tr>
<td>Viterbi</td>
<td>24</td>
<td>24</td>
<td>35</td>
<td>83</td>
</tr>
<tr>
<td>autocorr</td>
<td>12</td>
<td>0</td>
<td>10</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 6.5: 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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type I %</th>
<th>Type II %</th>
<th>Type III %</th>
<th>Total %</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>-</td>
<td>-</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>inSort</td>
<td>25</td>
<td>-</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>rle</td>
<td>-</td>
<td>100</td>
<td>84</td>
<td>91</td>
</tr>
<tr>
<td>tea8</td>
<td>58</td>
<td>75</td>
<td>100</td>
<td>68</td>
</tr>
<tr>
<td>Viterbi</td>
<td>92</td>
<td>83</td>
<td>80</td>
<td>84</td>
</tr>
<tr>
<td>autocorr</td>
<td>50</td>
<td>-</td>
<td>40</td>
<td>45</td>
</tr>
</tbody>
</table>

the MILU mutation testing tool [140] to generate “updates” corresponding to bug fixes. Table 6.4 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., \( A \lor B \rightarrow A \land B \)). 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 6.5 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 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 pro-
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 (Section 6.3.5). 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 ALUs 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.

6.5.4 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 [73, 74], 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 [77]). 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.

6.6 Related Work

6.6.1 Power Gating

In this chapter, we propose a method to reduce the area and power of applications running on a processor by removing unusable gates. Another

\footnote{Many embedded processors provide bare-metal development toolchains [75, 76].}
method to reduce power of unused gates is power gating. Prior work on aggressive power gating applies power gating at the granularity of RTL modules \([86, 102, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114]\). Some power gating techniques can even power down unused uncore modules (e.g., on-chip routers \([115, 116, 117]\)) or dynamically re-size microarchitectural structures to better suit an application \([86, 85]\). However, module-based power gating can only reduce power when an entire architectural or well-understood microarchitectural module is inactive, unlike our method for removing unusable gates, which can remove gates at a fine granularity. Solutions for fine-grained power gating also exist, that allow power gating to be performed at the gate or sub-module level. Unfortunately, fine-grained power gating incurs considerable area and power overheads (e.g., 40-50\% \([136]\)) for isolation gates, retention cells, and power switches. Therefore, fine-grained power gating is not a good fit for emerging area- and power-constrained applications.

We evaluated the effectiveness of aggressive module-based power gating for the same processor and benchmarks evaluated in Section 6.5. Figure 6.15 shows the maximum total power savings for an oracular, zero-overhead, module-based power gating technique, in which a module is assumed to dissipate no power in any cycle when none of the module’s gates are toggled. Additionally, no wakeup latency or energy is considered. Despite not including any of the overheads of power gating, the maximum power reduction for any application is less than 13\% – significantly lower than the minimum power re-

Figure 6.15: Power savings achieved by oracular power gating with no overheads are significantly lower than those achieved by bespoke processors for the same applications, even when each module is allowed a separate power domain and a wakeup latency of 0 is assumed.
duction provided by any of the bespoke processors for the same applications (37%). An actual power gating implementation would incur an area overhead for isolation cells, retention cells, and power switches. It would also incur latency overhead to wakeup modules when they are needed, which translates into a reduction in power savings and possibly reduced performance. In comparison, bespoke processors reduce power much more significantly than module-based power gating while incurring no performance overhead and also reducing design area.

Finally, it is worth noting that power gating and bespoke processor design are orthogonal and can be used together. For example, if a power gating technique is already applied to the baseline processor, our gate activity analysis will treat the power gating control logic and isolation cells like any other gates. If any do not toggle, the power gating control logic and isolation cells can be removed and replaced by the appropriate constant values. Power switch cells can be removed either if their control input is always constant or if their entire domain is cut. In this manner, bespoke design can be applied in conjunction with power gating to further reduce power.

6.6.2 High-Level Synthesis

High-Level Synthesis (HLS) tools such as Cadence Stratus [145] and Mentor Catapult [146] also aim to generate hardware for a given application. However, unlike bespoke processor design, HLS involves additional development cost since (a) a new high-level specification of application behavior needs to be defined, and (b) the high-level specification itself needs to be verified. Besides, while HLS tools can transform many C programs into efficient ASICs, there are well-known limitations that further increase development costs. Dynamic memory allocation, pointer ambiguity, extracting memory parallelism, and creating efficient schedules for arbitrary C programs are all challenges for HLS. In fact, most commercial tools limit the use of pointers and dynamic memory allocation, requiring additional hardware-aware design development to create a working ASIC from a C program. In contrast, our bespoke processor tool flow automatically creates a bespoke processor from the original, already-verified gate-level netlist and application binary without further design work. Also, unlike HLS, our bespoke tool flow can generate a
design that supports multiple applications on the same hardware (including full-fledged OS) and can support in-field updates. In these ways, a bespoke processor design flow can decrease design and verification effort and allow increased programmability compared with HLS tool flows.

6.6.3 Application- and Domain-Specific Cores

Recent work has studied the design of application- and domain-specific processors that improve energy-efficiency and increase performance by adding specialized hardware. Statically specialized cores, such as conservation cores [147], QsCores [148], and GreenDroid [149] automatically develop hardware implementations that are connected to a general-purpose processor at the data cache and target hotspots within an application code. Such cores increase energy efficiency at the expense of increasing the total area of a design, and thus may not be a good fit for area-constrained applications. Reconfigurable architectures, such as DySER [150], can also increase energy efficiency by mapping frequently executed code segments onto tightly coupled reconfigurable execution units. However, increased energy efficiency comes with an increase in area and power for the additional reconfigurable units. Extensible processors, such as Xtensa [142], allow a designer to specify configurations including structure sizing, optional modules (e.g., debug and exceptions), and custom application-specific functional units. Such extensible processors are limited in the extent to which they can reduce area and power, since they are applied primarily at the module level. Furthermore, the process is not fully automated and requires additional hardware design effort. Compared with extensible application-specific processors, bespoke processors can reduce power further, since they can remove gates within modules (see Section 6.5) and require less manual design effort.

Chip generators [151] can be used to generate families of chips from the ground up for a particular application domain by allowing domain expert hardware designers to encode domain-specific knowledge into tools that design application-specific chips within the same domain. Like HLS, this approach still requires a domain expert to design the overarching hardware in an HLS-like manner and then specify functions that allow arbitrary elaboration of the hardware design (e.g., encoding optimization functions for deter-
mining lower-level parameters such as cache associativity). Chip generators, therefore, require a change to the design process, while tailoring bespoke processors to applications can be completely automated from a program binary and processor netlist.

Simulate and eliminate\[152\] attempts to create a design tailored to an application by simulating the target application with a user-provided set of inputs on multiple base designs. Logic and interconnect components that are not used by the application are removed. Simulate and eliminate differs from bespoke processors in three fundamental ways – level of automation, scope of elimination, and correctness guarantees. First, simulate and eliminate requires significant user input to guide the selection of core parameters, selection of bit widths, and definition of optimizations. Bespoke processors require no user intervention. Second, simulate and eliminate only considers high-level, manually identified components when determining what is used by a processor, and consequently will not achieve as large of area and power reductions as fine-grained bespoke processor tailoring (Figure 6.12). Third, simulate and eliminate relies on user-specified inputs to determine the components that are never used by an application. This means that simulate and eliminate cannot guarantee safe optimization for applications where inputs affect control flow. Additionally, simulate and eliminate cannot determine if an unsafe elimination is performed. Bespoke processor tailoring guarantees correctness by considering all possible application inputs when determining which gates to remove.

6.7 Summary

In this chapter, we made a case for bespoke processors – processors that are tailored to a target application, such that they contain only the gates necessary to execute the application. We presented an automated methodology that takes a microprocessor IP and application as input and produces a bespoke processor with significantly lower area and power that is guaranteed to execute the application correctly for all possible executions and for all possible inputs. We showed that bespoke processors can have significantly lower area and power than their general-purpose counterparts, while maintaining support for multiple applications, as well as varying degrees of in-field pro-
grammability and debuggability. Average area and power reductions from bespoke processor design are 62% and 50%, respectively, while exploiting timing slack exposed by bespoke design improves average power savings to 65%.
Chapter 7

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

In the ultra-low-power embedded systems used by IoT applications, peak power and energy requirements are the primary factors that determine critical system cost characteristics, such as size, weight, cost, and lifetime. While the power and energy requirements of these systems tend to be application-specific, conventional techniques for rating peak power and energy cannot accurately bound the power and energy requirements of an application running on a processor, leading to over-provisioning that increases system size and weight. In this chapter, we present an automated technique that performs hardware-software co-analysis of the application and ultra-low-power processor in an embedded system to determine application-specific peak power and energy requirements. Our technique provides more accurate, tighter bounds than conventional techniques for determining peak power and energy requirements, reporting 15% lower peak power and 17% lower peak energy, on average, than a conventional approach based on profiling and guardbanding. Compared to an aggressive stressmark-based approach, our technique reports power and energy bounds that are both 26% lower, on average. Also, unlike conventional approaches, our technique reports guaranteed bounds on peak power and energy independent of an application’s input set. Tighter bounds on peak power and energy can be exploited to reduce system size, weight, and cost.

7.1 Introduction

Ultra-low-power (ULP) processors have rapidly become the most abundant type of processor in production today. New and emerging power- and energy-constrained applications such as the internet-of-things (IoT), wearables, implantables, and sensor networks have already caused production of ULP pro-
cessors to exceed that of personal computers and mobile processors [2]. The 2015 ITRS report projects that these applications will continue to rely on simple single-core ultra-low-power processors in the future, will be powered by batteries and energy harvesting, and will have even tighter peak power and energy constraints than the power- and energy-constrained ULP systems of today [82]. Unsurprisingly, low-power microcontrollers and microprocessors are projected to continue being the most widely used type of processor in the future [2, 3, 4, 137].

ULP systems can be classified into three types based on the way they are powered [153]. As illustrated in Figure 7.1, some ULP systems are powered directly by energy harvesting (Type 1), while some are battery-powered (Type 3). Another variant is powered by a battery and uses energy harvesting to charge the battery (Type 2).

For each of the above classes, the size of energy harvesting and/or storage components determine the form factor, size, and weight. Consider, for example, the wireless sensor node shown in Figure 7.2 [154]. The two largest system components that predominantly determine the overall system size and weight are the energy harvester (solar cell) and the battery.

Going one step further, since the energy harvesting and storage requirements of a ULP system are determined by its power and energy requirements, the peak power and energy requirements of a ULP system are the primary factors that determine critical system characteristics such as size, weight, cost, and lifetime [153]. In Type 1 systems, peak power is the primary constraint that determines system size, since the power delivered by harvesters...
is proportional to their size. In these systems, harvesters must be sized to provide enough power, even under peak load conditions. In Type 3 systems, peak power largely determines battery life, since it determines the effective battery capacity [155]. As the rate of discharge increases, effective battery capacity drops [155, 156]. This effect is particularly pronounced in ULP systems, where near-peak power is consumed for a short period of time, followed by a much longer period of low-power sleep, since pulsed loads with high peak current reduce effective capacity even more drastically than sustained current draw [156].

Figure 7.2: In most ULP systems, like this wireless sensor node, the size of the battery and/or energy harvester dominates the total system size.

In Type 2 and 3 systems, the peak energy requirement matters as well. For example, energy harvesters in Type 2 systems must be able to harvest more energy than the system consumes, on average. Similarly, battery life and effective capacity are dependent on energy consumption (i.e., average power) [156]. Figure 7.3 summarizes how peak power and energy requirements impact sizing parameters for the different classes of ULP systems. Finally, Tables 7.1 and 7.2 list the energy and power densities for different types of batteries and energy harvesters, respectively. These data provide a rough sense of how size and weight of a ULP system scale based on peak energy and power requirements. A tighter bound on the peak power and energy requirements of a ULP system can result in a roughly proportional
reduction in size and weight.

How Are Peak Power and Energy Determined Today?

There are several possible approaches to determine the peak power and energy requirements of a ULP processor (Figure 7.4). The most conservative approach involves using the processor design specifications provided in data sheets. These specifications characterize the peak power that can be consumed by the hardware at a given operating point and can be directly translated into a bound on peak power. This bound is conservative because it

---

Table 7.1: Specific energy and energy density for different battery types [157].

<table>
<thead>
<tr>
<th>Battery Type</th>
<th>Specific Energy [J/g]</th>
<th>Energy Density [MJ/L]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Li-ion</td>
<td>460</td>
<td>1.152</td>
</tr>
<tr>
<td>Alkaline</td>
<td>400</td>
<td>0.331</td>
</tr>
<tr>
<td>Carbon-zinc</td>
<td>130</td>
<td>1.080</td>
</tr>
<tr>
<td>Ni-MH</td>
<td>340</td>
<td>0.504</td>
</tr>
<tr>
<td>Ni-cad</td>
<td>140</td>
<td>0.828</td>
</tr>
<tr>
<td>Lead-acid</td>
<td>146</td>
<td>0.360</td>
</tr>
</tbody>
</table>
Table 7.2: Power density for different types of energy harvesters [158].

<table>
<thead>
<tr>
<th>Harvester type</th>
<th>Power Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photovoltaic (sun)</td>
<td>100 mW/cm²</td>
</tr>
<tr>
<td>Photovoltaic (indoor)</td>
<td>100 µW/cm²</td>
</tr>
<tr>
<td>Thermoelectric</td>
<td>60 µW/cm²</td>
</tr>
<tr>
<td>Ambient airflow</td>
<td>1 mW/cm²</td>
</tr>
</tbody>
</table>

Figure 7.4: The conventional methodology for sizing energy harvesting and storage components involves determining peak power and energy requirements for a processor and selecting components that will provide enough power and energy to satisfy the requirements over the lifetime of the system.
is not application-specific; however, it is safe for any application that might be executed on the hardware. A more aggressive technique for determining peak power or energy requirements is to use a peak power or energy stressmark. A stressmark is an application that attempts to activate the hardware in a way that maximizes peak power or energy. A stressmark may be less conservative than a design specification, since it may not be possible for an application to exercise all parts of the hardware at once. The most aggressive conventional technique for determining peak power or energy of a ULP processor is to perform application profiling on the processor by measuring power consumption while running the target application on the hardware. However, since profiling is performed with specific input sets under specific operating conditions, peak power or energy bounds determined by profiling might be exceeded during operation if application inputs or system operating conditions are different than during profiling. To ensure that the processor operates within its peak power and energy bounds, a guardband is applied to profiling-based results.

**Our Proposal: Determining Application-Specific Peak Power and Energy Requirements**

Most ULP embedded systems run the same application or computation over and over in a compute / sleep cycle for the entire lifetime of the system [65]. As such, the power and energy requirements of embedded ULP processors tend to be application-specific. This is not surprising, considering that different applications exercise different hardware components at different times, generating different application-specific loads and power profiles. For example, Figures 7.5a and 7.5b show the active (toggling) gates for two different applications (tHold and PI – see Table 7.3) during the cycles in which peak power is expended for each application. These figures were generated by running gate-level simulations of the applications on openMSP430 [63] and marking all gates that toggled in the cycle in which each benchmark expended its peak power. The figures show that PI exercises a larger fraction of the processor than tHold at its peak, leading to higher peak power. However, while the peak power and energy requirements of ULP processors tend to be application-specific, many conventional techniques for determining peak power and energy requirements for a processor are not application-specific (e.g., design-based and stressmark-based techniques). Even in the case of a profiling-based technique, guardbands must be used to inflate the peak
(a) Active gates at the peak cycle for tHold (452 gates).

(b) Active gates at the peak cycle for PI (743 gates).

Figure 7.5: Different applications can have different activity profiles, resulting in peak power and energy requirements that are application-specific.

power requirements observed during profiling, since it is not possible to generate bounds that are guaranteed for all possible input sets. These limitations prevent existing techniques from accurately bounding the power and energy requirements of an application running on a processor, leading to over-provisioning that increases system size and weight.

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

- We present an automated technique based on symbolic simulation that takes an embedded system’s application software and processor netlist as in-

\footnote{Peak power and energy analyses can be offered as a cloud compilation service by the hardware system vendor in settings where the application developer does not have access to the processor description [91, 92, 93].}
puts and determines application-specific peak power and energy requirements for the processor that are guaranteed to be valid for all possible application inputs and operating conditions. This is the first approach to use symbolic simulation to determine peak power and energy requirements for an application running on a processor.

- We show that the application-specific peak power and energy requirements determined by our technique are more accurate, and therefore less conservative, than those determined by conventional techniques. On average, the peak power requirements generated by our technique are 27%, 26%, and 15% lower than those generated based on design specifications, a stressmark, and profiling, respectively, and the peak energy requirements generated by our technique are 47%, 26%, and 17% lower. Reduction in the peak power and energy requirements of a ULP processor can be leveraged to improve critical system metrics such as size and weight.

- Our technique can be used to guide optimizations that target and reduce the peak power of a processor. Optimizations suggested by our technique reduce peak power by up to 10% for a set of embedded applications.

7.2 A Case for Application-Specific Input-Independent Peak Power and Energy Requirements

We measured peak power consumption for a sample set of ULP benchmark applications (see Table 7.3) running on an MSP430F1610 processor. Benchmark applications were run repeatedly with different inputs at an operating frequency of 8 MHz while sampling the voltage and current of the processor at a rate of 10 MHz using an InfiniiVision DSO-X 2024A oscilloscope, to ensure at least one sample per cycle. Power is calculated as the product of voltage and current. Figure 7.6 shows our test setup.

Figure 7.7a compares the peak power observed for different applications. The results show that peak power can be different for different applications. Thus, peak power bounds that are not application-specific will overestimate the peak power requirements of applications, leading to over-provisioning of energy harvesting and storage components that determine system size and weight. Figure 7.7a also shows that the peak power requirements of

---

3MSP430 is one of the most popular processors used in ULP systems [94, 121].
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 application depends on application inputs and can vary significantly for different inputs. This means that profiling cannot be relied on to accurately determine the peak power requirement for a processor, since not all input combinations can be profiled, and the peak power for an unprofiled input could be significantly higher than the peak power observed during profiling. Since input-induced variations change peak power by over 25% for these applications (Figure 7.7a), a profiling-based approach for determining peak power requirements should apply a guardband of at least 25% to the peak power observed during profiling.

For energy-constrained ULP systems, like those powered by batteries (Types 2 and 3), peak energy as well as peak power determines the size of energy harvesting and storage components (Section 7.1). Thus, it is also important to determine an accurate bound on the peak energy requirements of a ULP processor. Figure 7.8 shows the instantaneous power profile for an application (mult), demonstrating that on average, instantaneous power can
Figure 7.7: The peak power and normalized peak energy (normalized to an application’s runtime in cycles) of a ULP processor are different for different applications and different inputs. The bars represent average across all inputs; error bars show the range of input-induced peak and average power variations. Measured variation between multiple runs of the same application and same input is less than 2%.

be significantly lower than peak power. Therefore, we can more accurately determine the optimal sizing of components in an energy-constrained system by generating an accurate bound on peak energy, rather than conservatively multiplying peak power by execution time.

Figure 7.7b characterizes the peak energy, normalized to application runtime in cycles, for different applications and input sets, showing that the maximum rate at which an application can consume energy is also application- and input-dependent. Therefore, conventional techniques for determining the peak energy requirements of a ULP processor have the same limitations as conventional techniques for determining peak power requirements. In both cases, the limitations of conventional techniques require over-provisioning that can substantially increase system size and weight.

In the next section, we describe a novel technique for determining the peak power and peak energy requirements of a ULP processor that is application-specific yet also input-independent.
Figure 7.8: Measured instantaneous power of MSP430F1610 for the mult benchmark is significantly lower, on average, than both the rated and observed peak power for the application.

7.3 Determining Application-Specific Input-Independent Peak Power and Energy

Figure 7.9 provides an overview of our technique for determining application-specific peak power and energy requirements that are input-independent. The inputs to our technique are the application binary that runs on a ULP processor and the gate-level netlist of the ULP processor. The first phase of our technique, described in Section 7.3.1, is an activity analysis that uses symbolic simulation to efficiently characterize all possible gates that can be exercised for all possible execution paths of the application and all possible inputs. This analysis also reveals which gates can never be exercised by the application. Based on this analysis, we perform input-independent peak power (Section 7.3.2) and energy (Section 7.3.3) calculations to determine the peak power and energy requirements for a ULP processor.

7.3.1 Input-Independent Gate Activity Analysis

Since the peak power and energy requirements of an application can vary based on application inputs, a technique that determines application-specific peak power requirements must bound peak power for all possible inputs. Exhaustive profiling for all possible inputs is not possible for most applications, so we use the symbolic simulation-based hardware-software co-analysis presented in Chapter 4 that uses unknown logic values (Xs) for inputs to effi-
Figure 7.9: Our technique performs input-independent activity analysis that enables determination of accurate peak power and energy requirements for a ULP processor.

7.3.2 Input-Independent Peak Power Requirements

The input to the second phase of our technique is the symbolic execution tree generated by input-independent gate activity analysis. Algorithm 4 describes how to use the activity-annotated execution tree to generate peak power requirements for a ULP processor, application pair.

The first step in determining peak power from an execution tree produced during gate activity analysis is to concatenate the execution paths in the execution tree into a single execution trace. We use a value change dump (VCD) file to record the gate-level activity in the execution trace. The execution trace contains Xs, 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
Algorithm 4 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. foreach \{even—odd\} cycles \(c \in T\) do
5.  foreach toggled gates \(g \in c\) do
6.   if \(\text{value}(g,c) == X \&\& \text{value}(g,c-1) == X\) then
7.     \(\text{value}(g,c-1) ← \text{maxTransition}(g,1)\) // returns the value of the gate in the first cycle of the gate’s maximum power transition
8.   value(g,c) ← \text{maxTransition}(g,2) // returns the value of the gate in the second cycle of the gate’s maximum power transition
9.   else if \(\text{value}(g,c) == X\) then
10.  \(\text{value}(g,c) ← !\text{value}(g,c-1)\)
11.  else if \(\text{value}(g,c-1) == X\) then
12.   \(\text{value}(g,c-1) ← !\text{value}(g,c)\)
13.  end if
14. end for
15. \{E-O\}_VCD ← \text{value}(*,c-1)
16. \{E-O\}_VCD ← \text{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(\(P_{peak}\))

when the gate transitions (toggles). Since a transition involves two cycles, maximizing dynamic power in a particular cycle, \(c\), of the execution trace involves assigning values to any \(X\)s in the activity profiles of the current and previous cycles, \(c\) and \(c-1\), to maximize the number of transitions in cycle \(c\).

The number and power of transitions are maximized as follows. When the output value of a gate in only one of the cycles, \(c\) or \(c-1\), is an \(X\), the \(X\) is assigned the value that assumes that a transition happened in cycle \(c\). When both \(X\)s, the values are assigned to produce the transition that maximizes power in cycle \(c\). The maximum power transition is found by a look-up into the standard cell library for the gate. Since constraining \(X\)s in two consecutive cycles to maximize power in the second cycle may not maximize power in the first cycle, we produce two separate VCD files – one that maximizes power in all even cycles and one the maximizes power in all odd cycles. To find the peak power of the application, we first run activity-based power analysis on the design using the even and odd VCD files to generate even and odd power traces. We then form a peak power trace by interleaving the power values from the even cycles in the even power trace and the odd cycles in the odd power trace. This peak power trace bounds the peak power that is possible in every cycle of the execution trace. The peak power requirement of the application is the maximum per-cycle power
Figure 7.10: To determine a bound on peak power, we generate two different activity profiles – one that maximizes power in even cycles (left) and one that maximizes power in odd cycles (right).

Our VCD generation technique is illustrated in Figure 7.10. We use the example of three gates with overlapping Xs that need to be assigned to maximize power in every cycle. We show two assignments – one that 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.

7.3.3 Input-Independent Peak Energy Requirements

Our technique generates a per-cycle peak power trace characterizing all possible execution paths of an application. The peak power trace can be used to generate peak energy requirements. Figure 7.11 shows per-cycle peak power traces sampled from our benchmark applications. Since per-cycle peak power

---

4It is possible that glitching between clock edges can impact the power profile for an application. This impact can be accounted for by Primetime’s power analysis [126].
varies significantly over the compute phases of an application, peak energy can be significantly lower than assuming the maximum peak energy (i.e., \( \text{peak power} \times \text{clock period} \times \text{number of cycles} \)). Instead, the peak energy of an application is bounded by the execution path with the highest sum of per-cycle peak power multiplied by the clock period. To avoid enumerating all execution paths, we use several techniques. For an input-dependent branch, peak energy is computed by selecting the branch path with higher energy. For a loop whose number of iterations is input-independent, peak energy can be computed as the peak energy of one iteration multiplied by the number of iterations. For cases where the number of iterations is input-dependent, the maximum number of iterations may be determined either by static analysis or user input (as suggested by prior work [159]).\(^5\) If neither is possible, it may not be possible to compute the peak energy of the application; however, this is uncommon in embedded applications [65].

7.3.4 Validation of \(X\)-based Analysis

To demonstrate that our symbolic execution-based (\(X\)-based) activity analysis marks all gates that could possibly be toggled by an application for all possible inputs, we performed a validation check by comparing the sets of gates toggled by input-based simulations for several different input sets against the set of gates marked as potentially toggled by symbolic simulation. Figure 7.12 illustrates this comparison for two input-based simulations of the \textit{mult} benchmark with different input sets – those that have the lowest and highest number of toggled gates. In the figure, toggled gates common to \(X\)-based and input-based simulation are shown as \(X\)s, and gates that are exclusively marked by symbolic simulation as potentially toggled are shown as blue triangles. As expected, there are no gates that are exclusively marked by input-based simulation. Our validation results show that all the gates toggled by input-based simulation are also marked as potentially toggled by \(X\)-based symbolic simulation, validating the correctness of our approach for characterizing toggle activity.

We perform a second validation of our technique by comparing the peak power traces generated for benchmarks by our technique against power traces

\(^5\)The number of loop iterations is bounded for all evaluated benchmarks. In general, applications with unbounded runtimes are uncommon in embedded domains.
Figure 7.11: The per-cycle peak power varies significantly over the course of an application, showing that the worst-case average power can be significantly lower than peak power. Therefore, the peak energy can be significantly lower than the product of peak power and application runtime would suggest.
Figure 7.12: Toggled gates for \textit{mult} with low-activity inputs (top) and high-activity inputs (bottom), compared against potentially toggled gates identified by \textit{X}-based analysis. \textit{X}-based simulation marks all gates that can potentially toggle for an application for all possible inputs. This set of gates (\textit{unique\_x} $\cup$ \textit{common}) is a superset of the gates that toggle during an input-based application execution (\textit{common}).
Figure 7.13: The X-based peak power trace generated by our technique for an application provides an upper bound on all possible input-based power traces for the application (result shown for mult).

generated by input-based execution of the benchmarks. The validation results confirm that our peak power trace always provides an upper bound on the power of any input-based power trace. Figure 7.13 shows an example; the X-based peak power trace for the mult application is always higher than the input-based power trace. These validation results also show that the X-based peak power trace closely matches the input-based trace, indicating that the peak power and energy requirements generated by our technique are not overly conservative.

7.3.5 Enabling Peak Power Optimizations

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

Figure 7.14: A snapshot of instantaneous power profiles for `mult` at two different COIs where peaks occur. Our technique analyzes the instructions in the pipeline (top) to find each COI’s culprit instructions that cause the peak power in each pipeline stage along with the per-module peak power breakdown (bottom) to identify which instructions in which microarchitectural modules are responsible for a peak.

Figure 7.14 shows an example where our technique identifies peak power
spikes in cycles 146 and 150. Our technique also reports the instructions in each stage of the pipeline during those cycles of interest (COIs), as well as the per-module power breakdown for those cycles, which identifies the modules that are consuming the most power. This information can be used to guide optimizations that replace the instructions with different instruction sequences that induce less activity and power in the modules that consume the most power. Since software optimizations can impact performance as well as peak power, we will discuss optimizations that reduce peak power and their impact on performance and energy in Section 7.5.1.

7.4 Methodology

7.4.1 Simulation Infrastructure and Benchmarks

We verify our technique on a silicon-proven processor – openMSP430 [63], an open-source version of one of the most popular ULP processors [121, 94]. The processor is synthesized, placed, and routed in TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [122] and Cadence EDI System [123]. Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using a custom gate-level simulator that efficiently traverses the control flow graph of an application and captures input-independent activity profiles (Section 7.3). We show results for all benchmarks from [64] and all EEMBC benchmarks that fit in the program memory of the processor. These benchmarks are chosen to be representative of emerging ultra-low-power application domains such as wearables, internet of things, and sensor networks [64]. The IPC of these benchmarks on our processor varies from 1.25 to 1.39, with an average of 1.29. Power analysis is performed using Synopsys Primetime [126]. Experiments were performed on a server housing two Intel Xeon E-2640 processors (eight cores each, 2GHz operating frequency, 64GB RAM).

Section 7.2 shows measured data for an MSP430F1610 processor that demonstrate that different applications have different peak power and energy requirements, and the requirements of an application can vary significantly for different inputs. The results motivate an application-specific input-
independent technique for determining the peak power and energy requirements for ULP processors. For the results in Section 7.5, we perform evaluations on the open source openMSP430 processor [63]. Figures 7.15a and 7.15b confirm that the peak power and energy requirements of openMSP430 also depend on the application and application inputs. Note that the results in Figure 7.7 and Figure 7.15 differ because they are for different implementations of the MSP430 architecture (MSP430F1610 and openMSP430), with different process technology (130 nm vs. 65 nm) and operating frequencies (8MHz vs. 100 MHz).

Figure 7.15: Different applications and different input sets for the same application have different peak power and peak energy requirements (results for openMSP430).

7.4.2 Baselines

For baselines, we compare against conventional techniques for determining the peak power and energy requirements of processors. An overview of the baseline techniques can be found in Figure 7.4. The design specification-based baseline (design tool) is determined by performing power and energy analysis of the design using the default input toggle rate used by our design tools [126]. The stressmark-based baselines (GB input-based) use stressmarks that target peak instantaneous power and average power. Kim et al. used a genetic algorithm to automatically generate stressmarks that target maximum $di/dt$-induced voltage droop for a microprocessor [160]. We modi-
Table 7.3: Benchmarks.

<table>
<thead>
<tr>
<th>Embedded Sensor Benchmarks [64]</th>
<th>EEMBC Embedded Benchmarks [65]</th>
<th>Control Systems Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult, binSearch, tea8, intFilt, tHold, div, inSort, rle, intAVG</td>
<td>Autocorr, FFT, ConvEn, Viterbi</td>
<td>Proportional Integral Controller (PI)</td>
</tr>
</tbody>
</table>

field their framework to generate stressmarks that target peak instantaneous power and average power for openMSP430. The profiling-based baseline (input-based) is generated by performing input-based power and energy profiling for several input sets and applying a guardbanding factor of $4/3$ to the peak power and energy observed during profiling. The guardbanding factor is the same as in prior studies [161, 162] and is appropriate for the input-dependent peak power variability exhibited by our benchmarks (Figure 7.7a).

7.5 Results

We use our technique described in Section 7.3 to determine peak power and energy requirements for a ULP processor for different benchmark applications. Figure 7.16 compares the peak power requirements reported by our technique against the conventional techniques for determining peak power requirements, described in Section 7.4.2. The results show that the peak power requirements reported by our X-based technique are higher than the highest input-based application-specific peak power for all applications, confirming that our technique provides a bound on peak power. The results also show that our technique provides the most accurate bound on peak power, compared to conventional techniques for determining peak power requirements. For example, the peak power requirements reported by our technique are only 1% higher than the highest observed input-based peak power for the benchmark applications, on average. Other techniques for determining peak power and energy requirements are significantly less accurate, which can lead to inefficiency in critical system parameters such as size and weight (see Section 7.1).

Our technique is more accurate than application-oblivious techniques such
Figure 7.16: Our X-based technique for determining peak power requirements provides the most accurate (least conservative) guaranteed bound on peak power.

as determining peak power requirements from a stressmark or design specification, because an application constrains which parts of the processor can be exercised in a particular cycle. Our technique also provides a more accurate bound than a guardbanded input-based peak power requirement, because it does not require a guardband to account for the non-determinism of input-based profiling (shown in Figure 7.16 as error bars). By accounting for all possible inputs using symbolic simulation, our technique can bound peak power and energy for all possible application executions without guardbanding. The peak power requirements reported by our technique are 15% lower than guardbanded application-specific requirements, 26% lower than guardbanded stressmark-based requirements, and 27% lower than design specification-based requirements, on average.

Since our technique is application-specific and does not require guardbands, one question is: "Why is the bound provided by X-based analysis
more conservative for some applications than others?" The answer is that since $X$-based analysis provides a bound on power for all possible inputs, it becomes more conservative when there is greater possibility for input-dependent variation in power. For example, the multiplier is a relatively large, high-power module, with high potential for input-dependent variation in power consumption. For some inputs (e.g., $X \times 0$), 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 $X$s for inputs, we always assume the 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 applications, even those with more potential for input-induced power variation, our $X$-based analysis technique provides a peak power bound that is more accurate than those provided by conventional techniques.

Our technique also provides more accurate bounds on peak energy than conventional techniques, partly because of the reasons mentioned above, and also because our technique is able to characterize the peak energy consumption in each cycle of execution, generating a peak energy trace that accounts for dynamic variations in energy consumption. Using a design specification to determine peak energy is particularly inaccurate, since it does not consider dynamic variations in the energy requirements of an application. The guardbanded input-based technique, which does consider dynamic variations, provides a more accurate peak energy bound than the design specification for all benchmarks. However, it does not always provide a more accurate bound than the design specification for peak power, since peak power is an instantaneous phenomenon that is less dependent on dynamic variations. Figure 7.17 presents peak energy of different benchmarks, normalized to application runtime in cycles, i.e., peak average power, which characterizes the maximum rate at which the application can consume energy. In Figure 7.17, the peak energy requirements reported by our technique are
Figure 7.17: Our X-based technique for determining peak energy requirement (normalized to application run-time in cycles, i.e., the peak average power) is more accurate than existing conventional techniques. 17% lower than guardbanded application-specific requirements, 26% lower than guardbanded stressmark-based requirements, and 47% lower than design specification-based requirements, on average. As expected, application-specific normalized peak energy (Figure 7.17) varies less than peak power (Figure 7.16), since peak energy characterizes average peak power over the entire execution of an application, whereas peak power corresponds to one instant in the application’s execution.

As described in Section 7.1, more accurate peak power and energy requirements can be leveraged to reduce critical ULP system parameters like size and weight. For example, reduction in a Type 1 system’s peak power requirements allows a smaller energy harvester to be used. System size is roughly proportional to harvester size in Type 1 systems. In Type 2 systems, it is the peak energy requirement that determines the harvester size; reduction in
peak energy requirement reduces system size roughly proportionally. Since required battery capacity depends on a system’s peak energy requirement, and effective battery capacity depends on the peak power requirement, reductions in peak power and energy requirements both reduce battery size for Type 2 and 3 systems.

A ULP system may contain other components, such as transmitter/receiver, ADC, DAC, and sensor(s), along with the processor. All of these components may contribute to the system’s peak power and energy, and hence, the sizing of the harvester and battery. Tables 7.4 and 7.5 show the percentage reduction in the harvester size and battery size, respectively, from our technique for different fractions representing the processor’s contribution to the system’s peak power and energy. For a real system such as the one shown in Figure 7.2, which has a harvester area of 32.6cm$^2$ and a battery volume of 6.95mm$^3$, the area reduction of the harvester is 4.87, 8.44, or 8.75cm$^2$ if the system is designed using guardbanded input-based profiling, guardbanded stressmark, or design tool, respectively, for estimating the peak power of the processor. Similarly, the volume reduction of the battery is 0.42, 0.63, or 1.12mm$^3$, respectively.\(^6\) As expected, savings from our technique are higher when the processor is the dominant consumer of power and energy in the overall system.\(^7\)

\(^6\)The battery is a thin film battery of dimensions 5.7mm \(\times\) 6.1mm \(\times\) 200 \(\mu\)m (area of 34.7mm$^2$). Assuming the height of the battery does not change, the corresponding savings in battery area are 6.07, 9.01, and 16.18mm$^2$, respectively.

\(^7\)ITRS 2015 projections show that the microcontroller will be the dominant consumer of power in future IoT and IoE systems [82].
Table 7.5: Percentage reduction in battery volume compared to different baseline techniques, averaged over all benchmarks, for different percentage contributions of the processor energy to the overall energy of the system.

<table>
<thead>
<tr>
<th>Baseline</th>
<th>10%</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>GB-Input</td>
<td>1.74</td>
<td>4.37</td>
<td>8.74</td>
<td>13.11</td>
<td>15.73</td>
<td>17.48</td>
</tr>
<tr>
<td>GB-Stress</td>
<td>2.59</td>
<td>6.49</td>
<td>12.98</td>
<td>19.48</td>
<td>23.37</td>
<td>25.97</td>
</tr>
<tr>
<td>Design Tool</td>
<td>4.66</td>
<td>11.66</td>
<td>23.32</td>
<td>34.98</td>
<td>41.97</td>
<td>46.64</td>
</tr>
</tbody>
</table>

7.5.1 Optimizations

As discussed in Section 7.3.5, our technique can be used to guide application-level optimizations that reduce peak power. Here, we discuss three software optimizations, suggested by our technique, that we applied to the benchmark applications to reduce peak power. The optimizations were derived by analyzing the processor’s behavior during the cycles of peak power consumption. This analysis involves (a) identifying instructions in the pipeline at the peak, and (b) identifying the power contributions of the microarchitectural modules to the peak power to determine which modules contribute the most.

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

The second optimization aims to reduce the instantaneous activity in a peak cycle by delaying the activation of one or more modules, previously activated in a peak cycle, until a later cycle. For this optimization, we focus on the POP instruction, since it generates peaks in some benchmarks. The peaks are caused 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 7.18.

- **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 MOVed to the inputs of the multiplier and then the output is MOVed back to the processor. For a two-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 7.19: 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 7.19 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 (see Section 7.1). Our results show that peak power can be reduced by up to 10%, and 5% on average. Peak power dynamic range can be reduced by up to 34%, and 18% on average. Figure 7.20 shows the peak power traces for an example application before and after optimization, demonstrating that optimization can reduce the peak power requirements for an application.

Since optimizations that reduce peak power can increase the number of instructions executed by an application, we evaluated the performance and energy impact of the optimizations. Figure 7.21 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 7.3).

7.6 Related Work

Peak power has been analyzed in several settings in literature. In particular, several techniques have been proposed to estimate the peak power of a design. Hsiao et al. [163, 164] propose a genetic algorithm-based estimation of peak power for a circuit. Wang and Roy [165] use an automatic test generation technique to compute lower and upper bounds for maximum power dissipation for a VLSI circuit. Sambamurthy et al. [166] propose a technique that uses a bounded model checker to estimate peak dynamic power at the module-level. The technique is also functionally valid at the processor level. Najeeb et al. [167] propose a technique that converts a circuit behavioral model to an integer constraint model and employs an integer constraint solver to generate a power virus that can be used to estimate the peak power of the processor. To the best of our knowledge, no prior work exists on determining application-specific peak power for a processor based on symbolic simulation.

The above techniques require a low-level description of the processor (behavioral or gate-level). Techniques have also been proposed at the architecture-
level to predict when power exceeds the peak power budget or to lower the peak-to-average power variation. Sartori and Kumar [168] propose the use of DVFS techniques to manage peak power in a multi-core system. Kontorinis et al. [162] proposed a configurable core to meet peak power constraints with minimal impact on performance. Our technique identifies the peak power and energy requirements of a processor through hardware-software co-analysis.

Estimating peak energy of an application has been previously studied as the worst case energy consumption (WCEC) problem [159, 80, 169]. However, prior techniques do not use accurate power models, instead relying on microarchitectural models, which do not consider the detailed state of a processor or input values. As observed by [170], the power of an instruction can differ based on the previous instructions in the pipeline and its operand values. Our peak power computation technique analyzes an application on a gate-level processor netlist, allowing us to account for the fine-grained interaction between instructions and the worst-case operand values. The result is an accurate power model that can be used for WCEC analyses such as the
example analysis in Section 7.5. Prior work on worst-case timing analysis simply identified the timing-critical path through the program. However, the timing-critical path through a program may not be energy-critical [159, 169]. We calculate energy across all paths through gate-level simulation to determine the path with highest energy.

7.7 Summary

In this chapter, we showed that peak power and energy requirements for an ultra-low-power embedded processor can be application-specific as well as input-specific. This renders profiling methods to determine the peak power and energy of ULP processors ineffective, unless conservative guardbands are applied, increasing system size and weight. We presented an automated technique based on symbolic simulation that determines a more aggressive peak power and energy requirement for a ULP processor for a given application. We show that the application-specific peak power and energy requirements determined by our technique are more accurate, and therefore less conservative, than those determined by conventional techniques. On average, the peak power requirements determined by our technique are 27%, 26%, and 15% lower than those generated based on design specifications, a stressmark, and profiling, respectively. Peak energy requirements generated by our technique are 47%, 26%, and 17% lower, on average, than those generated based on design specifications, a stressmark, and profiling, respectively. We also show that our technique can be used to guide optimizations that target and reduce the peak power of a processor. Optimizations suggested by our technique reduce peak power by up to 10% for a set of benchmarks.
Chapter 8

Software-Based Gate-Level Information Flow Security for IoT Systems

The growing movement to connect literally everything to the internet (i.e., IoT) through ultra-low-power embedded microprocessors poses a critical challenge for information security. Gate-level tracking of information flows has been proposed to guarantee information flow security in computer systems. However, such solutions rely on non-commodity, secure-by-design processors whose requisite redesign and remanufacture are at odds with the cost requirements of IoT. In this chapter, we observe that the need for secure-by-design processors arises because previous works on gate-level information flow tracking assume no knowledge of the application running in a system. Since IoT systems typically run a single application over and over for the lifetime of the system, we see a unique opportunity to provide application-specific gate-level information flow security for IoT systems. We develop a gate-level symbolic analysis framework that uses knowledge of the application running in a system to efficiently identify all possible information flow security vulnerabilities for the system. We leverage this information to provide security guarantees on commodity processors. We also show that security vulnerabilities identified by our analysis framework can be eliminated through software modifications at 15% energy overhead, on average, obviating the need for secure-by-design hardware. Our framework also allows us to identify and eliminate only the vulnerabilities that an application is prone to, reducing the cost of information flow security by \(3.3\times\) compared to a software-based approach that assumes no application knowledge.

8.1 Introduction

Wearables, sensors, and the internet of things (IoT) arguably represent the next frontier of computing. On one hand, they are characterized by ex-
tremely low power and cost requirements. On the other hand, they pose a dire security and privacy risk. As the internet of things progresses toward the internet of everything, where nearly everything is connected to the internet via an embedded ultra-low-power processor, higher connectedness implies more security attack vectors and a larger attack surface. Similarly, immersive usage models imply newer, more sinister consequences. The security and privacy concerns are not theoretical either. In the last couple of years, reported IoT attacks include compromising baby monitors to enable unauthorized live feeds [171], interconnected cars to control a car in motion [172], smartwatches and fitness trackers to steal private information and health data [173], power grids and steel mills to render them offline [174], and medical devices with detrimental, perhaps fatal, consequences on patients’ health [175]. Consequently, security and privacy need to be first order design concerns for IoT systems.

Information-flow security is one of the most well-studied approaches to providing security and privacy in computer systems [176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187]. The goal is to track flows of information through a computer system and either detect or prevent illicit information flows between tainted (e.g., untrusted or secure) state and untainted (e.g., trusted or non-secure) state. Tracking and managing flows allows a computer system to support different information flow policies and provide information flow guarantees that security and privacy constructs and protocols can be built upon. An information flow security-based approach can be invaluable in context of IoT systems due to the above discussed security and privacy risks associated with such systems.

The vast majority of techniques for tracking and managing information flows operate at the level of the ISA or above. While these techniques allow tracking and management of explicit information channels, they are largely incapable of doing the same for implicit or covert channels (including timing channels) [186]. Gate-level information flow security approaches [186, 187] have been proposed to allow tracking and management of information flow channels at the finest-grained digital level – gates. These approaches typically augment hardware logic blocks with gate-level information flow tracking (GLIFT) logic to perform information tracking. They also specify a method for performing compositions of augmented logic blocks. A gate-level approach allows tracking of all information flows – implicit, explicit, and covert – al-
allowing one to build secure-by-design systems [188, 189] with varying degrees of programmability, performance, and area.

While gate-level approaches are effective at providing information flow security, such approaches require hardware modifications. For example, required hardware support may include stacks of isolated timers that can reset the PC, new pipeline control hardware, support to manage memory bounds masking, and partitioned memory structures (caches, branch predictors, etc.) [189]. While these modifications may be acceptable for certain high-assurance systems [189], the ultra-low cost requirements of many IoT applications and the volume nature of their microcontrollers may prohibit such modifications.

We observe that many of the required architectural changes arise because prior works assume that all software besides the kernel is completely unknown. Since many emerging IoT applications run the same software again and again for the lifetime of the system, we argue that there is a unique opportunity to build low-overhead gate-level information flow techniques for these IoT systems. Many IoT systems – consider wearables, implantables, industrial controllers, sensor nodes, etc. – perform the same task (or a small set of tasks) repetitively. However, cost reasons dictate that these systems are implemented using a programmable microcontroller running application software instead of an ASIC. We observe that for such systems, it may be possible for a commodity microcontroller to guarantee gate-level information flow security for a given application, even if a guarantee cannot be provided for all applications.1 Similarly, for some applications where gate-level information flow guarantees are not met, it may be possible to guarantee gate-level information flow security only through minimal changes to the application software, even if these changes will be inadequate at providing guarantees for all applications (or for other processors). The ability to guarantee gate-level information flow security for the applications of interest on commodity hardware, even if no guarantee is provided for all applications, allows trusted IoT execution without the programmability, performance, and monetary costs of specialized secure-by-design systems derived from previous gate-level approaches.

1In this dissertation, we refer to the application as the entire binary code loaded into a system’s program memory. This includes all computational tasks as well as all system software.
We rely on these observations to build a software tool that performs gate-level information flow tracking for a given application on a given processor design without any hardware design effort. The tool takes as input the processor’s gate-level netlist, unmodified application binary, and information flow policies of interest, and performs symbolic (i.e., input-agnostic) gate-level simulation of the application binary on the netlist to determine if any of the information flow policies could be violated. If none of the information flow policies could be violated at the gate-level, the processor is declared to guarantee gate-level information flow security for the application. If an information flow policy could be violated, the tool reports the offending instruction(s) to the programmer as warnings or errors. This information can then be used by the programmer or the compiler to modify application software such that gate-level information flow guarantees are met for the application. This analysis can be applied to an arbitrary application and even for a commodity hardware design. Also, our approach can be used selectively for the applications limiting overheads only to certain applications that need software modification.

This chapter makes the following contributions:

- We present the first approach to track gate-level information on a per-application basis. Our approach is software-based and can track gate-level information even on a commodity hardware design.
- We show that feedback from our application-specific gate-level information flow tracking can be used to modify application software in a way that guarantees information flow security.
- We show that application-specific software modification to prevent only the insecure information flows that a system is vulnerable to can reduce overheads significantly (i.e., by $3.3 \times$), on average, compared to an “always on” software-based approach that assumes no knowledge of the application running in a system.

8.2 Background and Related Work

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
Figure 8.1: 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). Taint is propagated through a gate based on the gate’s input values (e.g., $A, B$) and their corresponding taints (e.g., $A_T, B_T$). If a tainted input can affect the output (e.g., when $A = 0$, $A_T = 1$, and $B = 1$), the output becomes tainted. However, a taint does not propagate to the output when a tainted input cannot affect the output (e.g., due to masking, as when $A = 1$, $A_T = 1$, $B = 0$, and $B_T = 0$).

an information flow policy. Past work [176, 177, 178, 179, 180, 181] 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 [182, 183, 184, 185] 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 [186]. 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. [186] proposed gate-level information flow tracking (GLIFT). As shown in Figure 8.1, GLIFT augments 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 $3\times$ overhead in hardware. More recently, GLIFT has been used to statically track information flows [189]. 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 [186] to processors that can handle arbitrary computations through hardware compartmentalization [188, 189]. 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., [186] requires all loops to be statically bounded while [188] 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 the context of IoT systems, given the huge diversity of IoT systems being driven by commodity microcontrollers [138]. In this chapter, we design full systems that ensure the same non-interference policy as [189] (i.e., no un-
trusted 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 [190, 191, 192, 193]. 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 [193], 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.

8.3 Motivation

In this section, we motivate an application-specific approach for gate-level information flow security in IoT systems through a series of examples. In the first example (Figure 8.2), we consider existing secure-by-design systems based on gate-level information flow tracking. These systems have been designed assuming that the application software running on the system is unknown [186, 188, 189]. Figure 8.2 depicts a processor running an unknown application (all $X$s). In the figure, port P1 is an input port through which the processor may read tainted data. Also, a partition in the data memory is marked as containing tainted data. Since the application is unknown, we are forced to assume that the unknown instructions may read tainted data from all possible sources, propagate tainted data to all parts of the processor, and also write tainted data to all untainted ports and memory regions. I.e., we must assume that an unknown application has the potential to cause all possible information flow security violations. Faced with this possibility, the only way to guarantee information flow security is to design a secure-by-design system that includes hardware mechanisms to proactively prevent all possible insecure information flows. While this approach results in a system that is immune to all possible security violations that an arbitrary application may cause, such stringent security measures require modifications to processor hardware and may often be overly-conservative in an IoT system that runs a single, and often simple, application. For example, consider Fig-
Figure 8.2: Assuming that an application is unknown means that the application may perform any action, including reading from all possible sources of tainted data, propagating tainted data to all parts of the processor, and writing tainted data through all untainted ports if hardware-based mechanisms are not put in place to prevent insecure information flows.

Figure 8.3, which shows the same processor running a known application. When the application is run on the processor, it never writes tainted data to untainted ports or memory partitions. Therefore, it is possible to guarantee information flow security for this system without making any changes to the hardware or software. This example demonstrates that guaranteeing information flow security is possible, even for an application running on a commodity processor, when the application software is known. This is encouraging for security-critical IoT systems, which, due to economic considerations, more often than not rely on lightweight commodity processors.

For the next example, consider Figure 8.4, which shows the same processor running a different known application that reads an input from a tainted port and uses it as a base pointer (offset) to access data memory. Since the input is tainted, it is possible that the memory address calculated from the offset maps to the untainted region of memory, allowing tainted data to propagate through the memory to an untainted output port. Thus, the application contains a potential information flow security violation that could be either

---

2Tracking of information flow violations is simplified for demonstrative purposes in these examples by assuming that the only flows that exist are the ones visible in the abstract processor representation; actual identification of tainted information flows requires gate-level tracking to ensure complete coverage [186] (see Section 8.4).
exploited intentionally (e.g., an input supplied by a malicious attacker) or exposed unintentionally (e.g., an unfortunate input supplied by an unwitting user).

Although the application in Figure 8.4 is vulnerable to an insecure information flow, it does not necessarily mean that the application must be run on a secure-by-design system with hardware-based security mechanisms to ensure information flow security. Consider Figure 8.5, which shows a different, functionally equivalent version of the same application running on the same commodity processor. In this version of the application, the base address (offset) read from the tainted port is filtered through a masking operation that sets certain bits in the address (e.g., the most significant bits) to ensure that addresses computed using the offset map only to the tainted region of memory. Since this software change prevents the possibility of propagating tainted data to an untainted output port, no information flow security violations are possible for the modified application. Thus, through knowledge of the application and its potential security exploits, it is possible in this case to prevent information flow security violations in a system only by making changes to the software running in the system.

The examples in this section show that (1) it is possible to guarantee information flow security on a commodity processor without the use of restrictive,
Figure 8.4: This application is vulnerable to information flows that could jeopardize system security. The application uses tainted input data to compute the address for a write operation. The write taints untainted memory, allowing a violation when tainted data are sent out of an untainted port.

Figure 8.5: A simple change to the application in Figure 8.4 (masking the tainted memory address to limit its scope) renders the system immune to insecure information flows, demonstrating that it may be possible to provide information flow security on a commodity processor by changing the software that runs on the processor.
hardware-based information flow control mechanisms, and (2) it is possible to eliminate information flow security violations in an embedded system simply by making software modifications. However, these possibilities can only be realized with (1) knowledge of the application running in the system, and (2) a means of identifying all possible insecure information flows to which the application is vulnerable.

Based on these insights, we propose an application-specific approach to guaranteeing information flow security for IoT systems that identifies all information flow security violations that are possible for a system consisting of a commodity processor and application software and provides software-based techniques that can be used to prevent these information flow security violations.

8.4 Application-Specific Gate-Level Information Flow Tracking

Section 8.3 motivates the potential benefits of a software-based application-specific approach to information flow security, but bringing the application into the picture presents several challenges for gate-level information flow tracking. 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 8.6 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 run-
ning 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.

8.4.1 Input-Independent Gate-Level Taint Tracking

Algorithm 5 describes our input-independent gate-level taint tracking that is built on top of the symbolic co-analysis described in Chapter 4. Initially, the values of all memory cells and gates are set as unknown values (i.e., $X$s) and are marked as untainted. The system binary, consisting of both tainted and untainted partitions,$^3$ 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 8.2). 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 [186] and exemplified in Figure 8.1.

A key difference between our input-independent gate-level taint tracking and prior analyses such as $^\ast$-logic occurs when an unknown symbol propagates to the PC. For example, directly applying a $^\ast$-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 (i.e., the abstract representation of the PC is made concrete.

$^3$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.
while still retaining the taint values), 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 conservatively maintaining correctness in exploring possible execution states. Our scalable approach 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 $X$s 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 $X$s, and simulation continues from the conservative state.

The result of the conservative approximation technique is a pruned execution tree that stores both the logical and taint values at each point. Once a state, such as $S_2$, is observed for a second time, there is no further exploration down that path since all further states have already been considered. This conservative approximation technique allows input-independent gate-level taint tracking to complete in a tractable amount of time, even for applications with an exponentially large or infinite number of execution paths.\footnote{Some complex applications and processors might still require heuristics for exploration of a large number of execution paths [71, 72]; however, our approach is adequate for ultra-low-power systems, representative of an increasing number of future uses which tend to have simple processors and applications [82, 4]. For example, complete analysis of our most complex system takes 3 hours.}

### 8.4.2 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
Figure 8.6: 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.

Algorithm 5 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 $U!=$ ∅ do
12. $e \leftarrow U$.pop()
13. while $e$.PC_next != X and !.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. $e'' \leftarrow T$.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. foreach $a \in$ 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
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 [189]: input and output ports are labeled as trusted or untrusted and, independently, as secret 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.

8.4.3 Illustrative Example

This section illustrates how application-specific gate-level information flow tracking works. Figure 8.7 depicts application-specific gate-level information flow tracking on an example portion of a processor circuit using a symbolic execution tree that identifies all information flows in all execution paths of an application.

Consider a small portion of a processor represented by the simple state machine in the top left of Figure 8.7 and implemented by the circuit in the bottom left of the figure. During application-specific gate-level information flow tracking of the application binary, the gate-level circuit is symbolically simulated using logical 1s, 0s, and Xs (i.e., unknown value symbols). Along with the values of each gate, a taint value is associated with each gate and is propagated according to the gate type and input values of the gate.
(taint values are shown with a light gray background). The right side of Figure 8.7 contains an example (abbreviated) symbolic execution tree that tracks taint values through all possible execution paths of an application during application-specific gate-level information flow tracking. In cycle 0, the circuit starts out in an unknown, yet untainted state (i.e., both \( S \) and \( In \) are \( X \)s while \( S_T \) and \( In_T \) are 0s). As a result of the untainted reset asserted in cycle 0, the circuit enters a known state, \( S = 0 \). Input \( In \) becomes an untainted 1 in cycle 1, resulting in the circuit transitioning to an untainted \( S = 1 \) state in cycle 2. After cycle 2, the PC (not shown) becomes an unknown value (\( X \)), so symbolic simulation is split into two paths. Since \( In \) is a tainted 0 in cycle 2 and propagates its taint to \( S' \), both branches start in a tainted state \( S = 1 \) in cycle 3. In cycle 3 of the left-hand path, \( In \), which is unknown and untainted is XORed with \( S \), which is tainted, and the circuit transitions into an unknown tainted state (\( S = X, S_T = 1 \)). In cycle 4 of the left-hand path, a tainted reset is asserted, which puts the circuit into a known state, \( S = 0 \). However, since the reset signal was tainted, the output state of the flip-flop remains tainted (\( S_T = 1 \)). This illustrates that a tainted reset signal will not untaint processor state elements. However, on the right-hand path, an untainted reset is asserted in cycle 4. This does reset the circuit into a known and untainted state (\( S = 0, S_T = 0 \)).

After tracking taints through every execution of the application, the execution tree characterizes all possible information flows for the application and can be used to identify all possible information flow violations. The specific conditions that we check for violations are described in Section 8.5.1.

8.5 Guaranteeing Information Flow Security for an Application

In this section, we describe software-based techniques that eliminate information flow security vulnerabilities in applications. Section 8.5.1 establishes conditions that are sufficient to guarantee information flow security, and Section 8.5.2 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 8.5.3, we verify that the software transformations achieve information flow security when run on a commodity
processor, and in Section 8.5.4, we prove that the transformations satisfy the sufficient conditions and ensure information flow security.

8.5.1 Sufficient Conditions for Guaranteeing 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 8.2. 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 con-
ditions 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 [186, 188, 189]. 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 8.2) 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).
8.5.2 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 [188], 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 8.8 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).\(^5\) 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 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

\(^5\)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 8.8: Untainted timer reset example: In the left-hand code listing, all instructions after address 0 are marked as tainted. Once the first tainted instruction is loaded, the PC quickly becomes tainted and the jump back to the untainted instruction at address 0 does not reset the PC to being untainted. However, by setting the watchdog timer during the untainted portion of the code (the right-hand code listing) and padding the tainted portion with nops, the PC can be reset to an untainted value in the untainted partition of code.
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:**

Figure 8.9 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 pos-
Figure 8.9: Memory mask example: In the left-hand code listing, the instruction at address 4 reads a tainted and unknown input. At address 6, the tainted input is used to calculate the address for a store at address 7. Since the address is both unknown and tainted, that store ends up tainting the whole data memory space. By adding two instructions at addresses 7 and 8 (see the right-hand code listing) that mask the address to only use the tainted task’s memory partition, no untainted memory locations become tainted.
sibility 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.

8.5.3 Verification of Software Techniques

Here, we verify that our software techniques for guaranteeing information flow security indeed work using the micro-benchmarks presented in Figure 8.8 and Figure 8.9 and an unmodified openMSP430 processor. Consider the left-hand code listing in Figure 8.8. We initialize the input-independent gate-level taint tracking such that the instructions shaded gray are tainted. During any possible execution of the application, once the PC becomes tainted, it never becomes untainted again. However, if the watchdog timer is set using untainted code (see the right-hand code listing in Figure 8.8), each execution of the untainted code section has a trusted PC. Now consider the right-hand code listing in Figure 8.9. Here, the code itself is not marked as tainted, but the code reads data from a tainted port and uses it to index into an array. During input-independent taint tracking, each input that is read from the tainted port is tainted. We observe during information flow tracking that the entire memory space becomes tainted, due to the propagation of tainted data to a memory address calculation. When instructions are inserted that guarantee that the unknown address is bounded to the tainted task’s region in data memory, then the result of information flow tracking indicates that no untainted memory locations can be tainted.

8.5.4 Proving Information Flow Security

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

Proof: For tainted data to influence the execution of $A_I$, a taint must
propagate from a tainted input of $S$ to an untainted output written by $A_T$ either through a state element of $P$, through the memory, or directly from a port.

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

**Case 2 – taint propagation through memory:** For taintedness to influence $A_T$ through the memory, a tainted computational task $A_J$ must write to some memory location $M$ outside its tainted memory partition, and $A_T$ must read from that memory location while it is executing on $P$. However, in any case where $T_S$ identifies that $A_J$ could write outside of its memory partition, $A$ is modified such that masking instructions are inserted to ensure that $A_J$ can only write inside its own memory partition. Furthermore, $T_S$ checks all possible execution states of $A$ on $P$ and reports that $A$ 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 $T_S$ means that it is impossible for tainted data to propagate through memory and influence the execution of $A_T$.

**Case 3 – taint propagation through a port:** For taintedness to propagate to an output through a port, either some $A_T$ must read from a tainted port or some $A_J$ must write directly to an untainted port. Both cases are reported as insecure by $T_S$ as it evaluates all possible execution states of $A$. Therefore, assurance of security from $T_S$ means that it is impossible for tainted data to influence the execution of $A_T$ or propagate to an untainted output from a port. 

■

209
Figure 8.10: Software refactoring tool flow: Based on the results of application-specific gate-level information flow tracking (Section 8.4), root cause identification generates a list of instructions that violate the untainted memory partitions condition and code tasks (e.g., functions) that violate the untainted control flow condition. These can then be used by a programmer or compiler to apply software-based fixes for the information flow violations.

8.6 A Toolflow for Software-Based Gate-Level Information Flow Security

We have developed an end-to-end toolflow, depicted in Figure 8.10, for developing systems that guarantee information flow security on commodity processors. The first stage in the toolflow checks whether an application conforms to a given information flow security policy. This stage takes as input the application software, including application code, library code, and system code, as well as the gate-level description of the processor, and performs application-specific gate-level information flow tracking (Section 8.4) on the system for a developer-defined information flow security policy that provides tainted / untainted labels for hardware and software (e.g., ports, code partitions, data partitions). The output of information flow tracking is a list of all possible information flow violations that may be generated by the application, along with cycle-accurate tainted state for each type of information flow.

To guarantee information flow security for the system, all identified violations must be eliminated by modifying the application software. To this end, the next stage of the toolflow reports potential information flow security violations to the developer at instruction-level granularity. This stage identifies the root cause of each potential gate-level violation – i.e., the instructions that lead to violations. For violations where the PC becomes tainted during
execution of a tainted code partition, our root cause identification tool marks the tainted partition as having tainted control flow, requiring the watchdog mechanism to be invoked. In cases where a store instruction in a tainted program partition can potentially write to an untainted memory partition, the static instruction (identified by its address in the program memory) is marked as needing masking.

The final stage of our toolflow refactors the software of the application in order to guarantee information flow security. The necessary software modifications identified by root cause analysis can be applied either manually or automatically by the compiler (Figure 8.11).\textsuperscript{6} For each instance where the compiler applies a modification to the software to eliminate a possible insecure information flow, it also reports a compile error or warning (depending on the severity of the violation) to the developer, indicating the line of code that caused the violation and the change that was made to fix the violation. Errors are reported for direct information leaks of tainted data that are not allowed (e.g., tainted code writes to an untainted output port), and warnings are reported for violations that may indirectly lead to information leaks if not fixed (e.g., a store from tainted code can write to an untainted memory partition). Reviewing the list of compile errors and warnings can be informative, since some violations and fixes are unavoidable (e.g., tainted control flow resulting from a control instruction that depends on a tainted input, which is fixed using the watchdog timer), while other violations may be caused by unintended software bugs (e.g., a store that is vulnerable to buffer overflow, which is fixed by masking). In the case of unintended software bugs, changing the program code may avoid the need for automated software modification to eliminate violations (e.g., fixing the buffer overflow problem avoids the need to mask the store).

After software has been modified to eliminate all possible information leaks of tainted data, application-specific gate-level information flow tracking can be used to verify that it is now impossible for the system to violate the

\textsuperscript{6}Two specific cases require programmer attention. First, if an untainted task directly accesses a tainted memory location or input port or a tainted task directly accesses an untainted output port, there is a fundamental violation in the software. In this case, an error is reported and the programmer must either change the software to eliminate the illegal access or redefine the information flow security labels. Second, if a tainted task originally uses the watchdog and also requires the watchdog for information flow guarantees, the programmer must either avoid using the watchdog or refactor the program to avoid tainting control flow (see Section 8.5.2 for details).
Figure 8.11: Automated software modification to eliminate information flow security violations for MSP430 involves C/C++ source code compilation and assembly into an object file that is linked with libraries, including a runtime support library to generate an executable object file, which is converted into hex that can be loaded into program memory. Application-specific information flow tracking and root-cause analysis use the final hex (program memory contents) to identify code tasks and store instructions that can cause violations. If a watchdog timer is needed, it is enabled in the system software via a #define. Any necessary mask insertions are performed in the assembly file at the specific addresses (instructions) identified by root cause analysis (if the watchdog was needed, analysis must be performed again on the new assembly file prior to mask insertion). The new, modified assembly file is run through the remainder of the toolflow to produce a new hex file to be loaded into the program memory.
specified information flow policy, i.e., the system now guarantees information flow security.

The feedback provided by our toolflow potentially represents another benefit of application-specific information flow tracking over secure-by-design processors. Our toolflow identifies and reports all possible causes of insecure information flows. Thus, security vulnerabilities are brought to the developer’s attention and can be addressed appropriately, resulting in an application that is secure. In a secure-by-design processor, hardware mechanisms are used to alter the functionality of the application silently, so an application’s security vulnerabilities may never be remedied, or even known. Also, violations corrected silently in hardware may manifest as runtime errors. For example, address masking performed by hardware can fix a buffer overflow problem, but the result is probably to map the store to some erroneous location inside the buffer, resulting in an erroneous execution / output for the application.

8.7 Results

**Processor and Benchmarks:** We perform evaluations on a silicon-proven processor – openMSP430 [63], an open-source version of one of the most popular ultra-low-power processors [121, 94]. The processor is synthesized, placed, and routed in TSMC 65GP technology (65nm) for an operating point of 1V and 100 MHz using Synopsys Design Compiler [122] and Cadence EDI System [123]. Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using a custom gate-level simulator that implements application-specific gate-level information flow tracking (Section 8.4). We show results for all benchmarks from [64] and all EEMBC benchmarks [65] that fit in the program memory of the processor (Table 8.1). Benchmarks are chosen to be representative of emerging ultra-low-power application domains such as wearables, internet of things, and sensor networks [64]. Benchmark performance (IPC) on our processor varies from 1.25 to 1.39.

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
### Table 8.1: Benchmarks.

<table>
<thead>
<tr>
<th>Embedded Sensor Benchmarks [64]</th>
<th>EEMBC Embedded Benchmarks [65]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult, binSearch, tea8, intFilt, tHold, div, inSort, rle, intAVG</td>
<td>Autocorr, FFT, ConvEn, Viterbi</td>
</tr>
</tbody>
</table>

restart the benchmark after each execution.

#### 8.7.1 Information Flow Violations

Application-specific gate-level information flow tracking reports all possible information flow violations for an application. Table 8.2 shows which of the unmodified benchmarks violated the sufficient conditions described in Section 8.5.1. 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.\(^7\) These benchmarks require the techniques described in Section 8.5.2 to guarantee information flow security. After performing software modifications identified by our toolflow, all condition violations are eliminated.\(^8\) 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.

\(^7\)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.

\(^8\)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 MSP430 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.
Table 8.2: Benchmarks that violate sufficient conditions 1 and 2 for information flow security (see Section 8.5.1) before and after modification.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No Mod</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>binSearch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>div</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>inSort</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>intAVG</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>intFilt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tHold</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tea8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Viterbi</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ConvEn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>autocorr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3: Performance overhead (%) for watchdog timer reset and memory address masking applied with and without application analysis.

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

8.7.2 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 8.3 (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 MSP430 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 inter-
vals and the overhead of state checkpointing and recovery (context switching)
for time slicing.\(^9\) 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 pre-
cisely which 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, guaran-
teeing information flow security for an unknown application requires masking
of every store and time bounding of every tainted task using a deter-
ministic timer, since all sufficient conditions must be satisfied to guarantee
non-interference, even though they may not be necessary for a particular ap-
lication (Section 8.5.1). Without the ability to identify all possible tainted
information flows for all possible executions of an application on a commod-
ity 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 8.3 (W/o App) shows the performance overhead
of using masking for all stores and time bounding for all tainted tasks, rep-
resenting a case where application analysis is not available and all sufficient
conditions must be enforced. In this case, performance overhead is \(3.3 \times\)
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, ap-
plying 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.

\(^9\)For openMSP430, the overhead of saving and restoring a task’s state is 20 cycles, and
watchdog timer initialization and reset takes 10 cycles.
8.7.3 Information Flow Secure Scheduling:  
A System-Level Use Case

In this section, we show that we can use the techniques developed in this chapter 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 [77] performs task scheduling between two tasks – \texttt{div} and \texttt{binSearch} – where \texttt{binSearch} is an untrusted task (its input and output ports are marked as untrusted), and FreeRTOS and \texttt{div} are both trusted.

The control flow of \texttt{binSearch} depends on an untrusted input value. Thus, in the baseline case, after \texttt{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 \texttt{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 use our toolflow to modify the system’s application, consisting of FreeRTOS and the two computational tasks. 330 store instructions in \texttt{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 \texttt{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.

8.8 Summary

IoT applications present a stress test for information flow security. The rapidly increasing quantity and variety of IoT devices also increases the quantity and variety of data handled by the devices, while driving down the power and area constraints, leaving little budget for security in this security-critical domain. In this chapter, we showed how knowledge of the application that runs on an IoT device can be leveraged to identify all possible information flow security vulnerabilities in the system, modify application software to eliminate vulnerabilities, and provide a guarantee that a system is information flow secure, even for systems built upon commodity ultra-low-power processors commonly used in IoT applications. Since our analysis framework identifies and eliminates only the information flows that a particular application is vulnerable to, the cost of eliminating all insecure information flows with our application-specific approach to information flow security is $3.3\times$ lower than a software-based approach that assumes no application knowledge.
Emerging applications in the IoT domain, such as wearables, implantables, smart tags, and wireless sensor networks put severe power, cost, reliability, and security constraints on hardware system design. Although ASICs can provide power efficiency, due to design and programmability costs, these IoT applications will be powered by ultra-low-powered microprocessors and microcontrollers. This dissertation focuses on the architecture and design of dependable ultra-low-power computing systems. Specifically, it proposes architecture and design techniques that exploit the unique application and usage characteristics of future computing systems to deliver low power, while meeting the reliability and security constraints of these systems.

First, this dissertation considers the challenge of achieving both low power and high reliability in SRAM memories. As the operating voltage of SRAMs is decreased to save power, the fraction of faulty cells rapidly increases, forcing a processor to operate at a higher voltage and thus higher power or use costly error correction to allow low voltage operation. This dissertation proposes both an architectural technique to reduce the overheads of error correction and a technique that uses the nature of error correcting codes to allow lower voltage operation at the same reliability target.

Second, this dissertation considers low power and low cost. By leveraging the fact that many IoT systems are embedded in nature and will run the same application for their entire lifetime, fine-grained usage characteristics of the hardware-software system can be determined at design time. This dissertation developed a novel hardware-software co-analysis based on symbolic simulation that can determine the possible states of the processor throughout any execution of a specific application. This enables power-gating where more gates are turned off for longer, bespoke processors customized to specific applications, and stricter determination of peak power bounds.

Third, this dissertation considers achieving secure IoT systems at low cost.
and power overhead. By leveraging the hardware-software co-analysis, this dissertation shows that gate-level information flow security guarantees can be provided without hardware overheads.

Although this dissertation made contributions toward addressing all four of the challenges for IoT processors, there is still work to be done. If many IoT applications are going to be powered by energy-harvesting devices, new architectural approaches will be needed that take into account the unique characteristics of an energy harvesting system. First, processors powered by energy harvesting systems are unlike current systems in that many such systems are not energy-limited, but are power limited. Second, the availability of power can change drastically over time. One direction for future work is developing low-power architectures to enable adaptive power-neutral computing. Such approach should be able to significantly improve the power-performance range that may be possible for future energy-harvesting systems.

While Bespoke processors provide power and area efficiency, they are limited in their programmability and provide limited to no performance benefit. In order to provide increased flexibility, design frameworks for building mostly general-purpose processors could be developed. For example, today if we have an instruction to be removed we do not know all of the hardware to be removed. Analyses similar to the hw-sw co-analysis can be developed to identify the components used by an instruction. In fact, such a framework can be cognizant of the costs and benefits of keeping an instruction in the ISA. The resulting automated hardware editing can be performed at other granularities as well.

Finally, since IoT systems will be involved in physical-facing applications where potential consequences of a systems behavior are serious, it may be critical to not only provide security, but also auditability. For example, being able to reconstruct the interactions between multiple IoT system. Providing such functionality at the very low cost required by IoT systems will be challenging. Using an analysis of both hardware and software together could help address these challenges.
References


226


