Exploiting Dynamic Timing Slack for Energy Efficiency in Ultra-Low-Power Embedded Systems

Hari Cherupalli
University of Minnesota
Minneapolis, Minnesota
Email: cheru007@umn.edu

Rakesh Kumar
University of Illinois
Urbana-Champaign, Illinois
Email: rakeshk@illinois.edu

John Sartori
University of Minnesota
Minneapolis, Minnesota
Email: jsartori@umn.edu

***Abstract*—Many emerging applications such as the internet of things, wearables, and sensor networks have ultra-low-power requirements. At the same time, cost and programmability considerations dictate that many of these applications will be powered by general purpose embedded microprocessors and microcontrollers, not ASICs. In this paper, we exploit a new opportunity for improving energy efficiency in ultra-low-power processors expected to drive these applications – dynamic timing slack. Dynamic timing slack exists when an embedded software application executed on a processor does not exercise the processor's static critical paths. In such scenarios, the longest path exercised by the application has additional timing slack which can be exploited for power savings at no performance cost by scaling down the processor's voltage at the same frequency until the longest exercised paths just meet timing constraints. Paths that cannot be exercised by an application can safely be allowed to violate timing constraints. We show that dynamic timing slack exists for many ultra-low-power applications and that exploiting dynamic timing slack can result in significant power savings for many ultra-low-power processors. We also present an automated methodology for identifying dynamic timing slack and selecting a safe operating point for a processor and a particular embedded software. Our approach for identifying and exploiting dynamic timing slack is non-speculative, requires no programmer intervention and little or no hardware support, and demonstrates potential power savings of up to 32%, 25% on average, over a range of embedded applications running on a common ultra-low-power processor, at no performance cost.**

I. Introduction

A large number of computing applications have recently exploded onto the scene. Notable among them include the internet of things, wearables, and sensor networks. A common thread running across these applications is that they often have ultra-low-power requirements [1], [2], [3], [4], [5]. These requirements are due to the fact that these applications are either energy-constrained (e.g., when they are battery powered) or power-constrained (e.g., applications with inductive coupling or power scavenging).

Considering that these applications tend to be embedded, one option to target these applications is to use an ASIC. However, many of these applications are cost-sensitive. Also, many of these applications need to be tuned or updated in the field. As a result, an ultra-low-power embedded microprocessor or microcontroller is often a better fit.

The question we ask in this research is: are there unique opportunities for power reduction for these emerging applications that exploit (a) the embedded nature of these applications and (b) the fact that these applications are often driven by ultra-low-power microprocessors and microcontrollers?

In this paper, we identify one such opportunity to reduce power consumption in ultra-low-power embedded systems without reducing performance. This opportunity leverages the observation that an embedded software application running on an ultra-low-power processor may not utilize all the functionalities provided by the processor. Only the parts of a processor that can be utilized by an application need to meet timing constraints. Therefore, in scenarios where unused functionalities correspond to timing-critical logic, there may exist timing slack between the most timing-critical functionalities in the processor and the most timing-critical functionalities that are *exercised* by the embedded software application running on the processor. We call this workload-dependent timing slack *dynamic timing slack* (DTS). We show in Section III that considerable DTS may exist for ultra-low-power processors.

Since only exercised parts of the processor need to meet timing constraints for the processor to work correctly, we can exploit DTS to improve the energy efficiency of the processor by operating the processor at a lower voltage for the same frequency. A safe, energy-efficient operating voltage ensures that all utilized functionalities of a processor meet timing constraints, while functionalities that cannot be toggled by an embedded software application are allowed to violate timing constraints. Nevertheless, unlike timing speculative approaches that save power by reducing safety guardbands (e.g., Razor [6]), exploiting DTS does not involve reducing guardbands for the subset of processor logic that is exercised by an application and therefore is *completely non-speculative*.

Exploiting DTS to improve energy efficiency of an ultra-low-power embedded system requires a methodology for identifying when DTS exists in a processor and how much DTS can be exploited while still guaranteeing safe operation of the processor. We present an automated technique for identifying and exploiting DTS that is non-speculative and requires no programmer intervention and little or no hardware support, making it particularly relevant for the ultra-low-power embedded systems we target in this work. The proposed methodology can be used to optimize the minimum voltage for the software running on an embedded ultra-low-power processor. It can also optionally be used to improve performance of the end application. We have packaged our DTS identification technique as an automated tool that takes an application binary and processor RTL as input and determines the minimum safe operating voltage for the application on the processor.[1].

Our paper makes the following contributions.

- We identify a novel opportunity to improve the energy efficiency of ultra-low-power embedded systems, based on identification and exploitation of *dynamic timing slack*.
- We present measurement-based evidence that considerable DTS exists in common ultra-low-power microprocessors.
- We present an automated technique for determining the amount of dynamic timing slack that can be exploited for a particular ultra-low-power processor running an embedded software application and

[1]Our automated DTS identification tool is available for download at the following link: http://www.ece.umn.edu/users/jsartori/tools.html

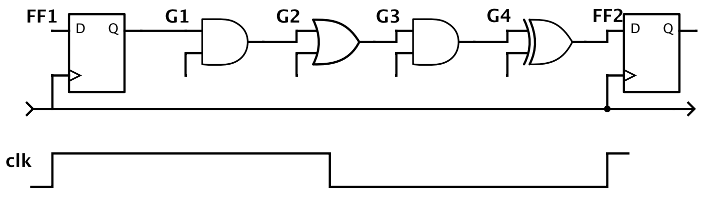


Fig. 1. A path in a synchronous circuit.

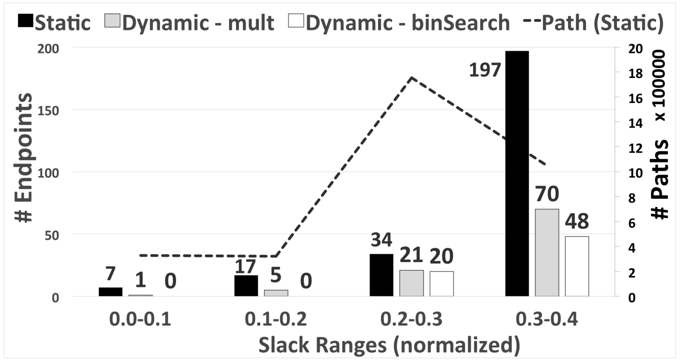


Fig. 2. Static slack distribution vs dynamic slack distributions for two applications (*mult* and *binSearch*) on openMSP430. Slack is normalized to the processor clock period. The dynamic slack distributions show that both applications do not exercise all of the endpoints in the processor.

the corresponding minimum operating voltage that ensures safe execution of the application on the processor, without any programmer intervention. To the best of our knowledge, this is the first known methodology that determines an application-specific V_{min} (or f_{max}) that is guaranteed to be safe irrespective of the input or operating conditions.

• We propose microarchitectural support to increase the benefits possible from exploiting dynamic timing slack.

• We show potential power savings of up to 32%, 25% on average, for a common ultra-low-power processor over a range of embedded applications, at no performance cost.

The rest of the paper is organized as follows. Section II explains the idea of workload-dependent DTS and motivates exploiting DTS for improved energy efficiency. Section III presents measurement-based observations of DTS in modern ultra-low-power processors. Section IV presents an automated methodology for identifying and exploiting DTS and discusses optional architectural support that can be used to increase the power benefits. Section V describes our experimental methodology in detail. Section VI presents experimental results that demonstrate significant power savings from exploiting DTS and compares DTS with related work. Section VII concludes the paper and discusses future work directions.

II. Workload-Dependent Dynamic Timing Slack

Most modern processors are synchronous or clocked. This means that computation is performed in clock periods where data is passed from flip-flop (FF) to flip-flop (launch FF or startpoint to capture FF or endpoint) through some combinational logic gates. Transmission from launch to capture FF begins with a clock signal to the launch FF and must complete before the next clock signal reaches the capture FF (i.e., one clock period). For example, in Figure 1 a logic transition (toggle) initiated by the data at the Q-pin of FF1 (launch FF) must reach the D-pin of FF2 (capture FF) in one clock period. A path that respects these constraints is said to meet timing. If the combined delay of gates G1 through G4 is greater than the clock period, the path does not meet timing. Decreasing the operating voltage reduces power consumption, but also increases logic delays, which can potentially cause a path to violate timing constraints. Note, however, that if the output of a path (e.g., the D-pin of FF2) does not toggle, *the circuit will still operate correctly even if the path violates timing constraints*, since the capture FF will still capture the correct (constant) value in each clock period.

A typical processor has a large number of paths [7], and all paths *must* meet timing. However, some paths may just meet timing with little time to spare (timing-critical paths), while for other paths the correct data is available at the D-pin of the capture FF (endpoint) significantly before the end of a clock period. We observe that many emerging applications for ultra-low-power processors do not utilize a GPP's entire feature set [1], [4], [5]. Non-exercised features can mean that only a subset of the paths in a processor are exercised (toggled). If the longest *exercised* path in the processor is not a timing-critical path (i.e., it produces data at its endpoint with time to spare), then there exists an opportunity to trade this extra time for reduced power *at no performance cost* by keeping frequency constant and reducing the voltage of the processor to the lowest voltage where all exercised paths still meet timing. Any un-exercised paths are allowed to violate timing constraints. This is the opportunity we exploit in this research. Remember, as mentioned above, that if a path is not exercised, its output will remain constant, and the capture FF for the path will continue to capture the correct value in each clock period.

We now demonstrate the opportunity described in this section with an example that illustrates the existence of DTS in an example ultra-low-power processor. Figure 2 compares several different slack distributions for slack up to 40% of the clock period, for a fully synthesized, placed, and routed openMSP430 processor [8]. In the figure, the x-axis has bins for various slack ranges (normalized to the clock period), the left y-axis shows the number of processor endpoints with worst slack in a particular range, and the right y-axis shows the number of paths with slack in a particular range.[2] The static slack distributions, *Static* and *Path (Static)*, characterize the worst slacks of all endpoints and paths in the processor, respectively, whether exercised or not. Note that a large number of paths in the design are statically critical (over 325000). This is consistent with previous observations on other designs [9], [10]. Nevertheless, when a particular application is executed on the processor, not all the paths or path endpoints may be exercised. The other two series in Figure 2 show the distributions of worst slacks for only the endpoints in the processor that are exercised by the *mult* and *binSearch* applications (see Table III). We call these *dynamic slack distributions*, and we call the longest exercised paths in a design the *dynamic critical paths* for a particular application [11]. Slack distributions are reported at the worst-case corner to isolate DTS from all other phenomena that might affect the minimum operating voltage for an application (e.g., voltage, temperature, or aging variations).

The following observations and inferences can be drawn from Figure 2.

• Several endpoints of the processor (and hence orders of magnitude

[2]Worst slack is defined for an endpoint (FF) as the timing slack of the longest path terminating at that endpoint. Since many paths lead to the same endpoint, the number of paths in a design is typically several orders of magnitude larger than the number of endpoints [7]. In our processor, each endpoint corresponds to tens or hundreds of thousands of paths.

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

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

• DTS represents an opportunity to *save power without sacrificing performance*. For example, if *binSearch* is executing on the processor, operating voltage can be reduced while keeping frequency constant, such that paths with timing slacks of up to 20% of the clock period of the processor violate timing constraints (since these paths are not exercised by the application). This generates power savings (see Section VI) without affecting either the functionality or performance of the processor for *binSearch*. Note also that unlike timing speculative approaches that save power by reducing safety guardbands (e.g., Razor [12], [13], [6]), exploiting DTS does not require guardband reduction and therefore is *completely non-speculative*. Exploiting DTS simply involves adjusting the voltage of the processor to the minimum *safe* voltage for the subset of processor logic that is exercised by an application. Guardbands for the exercised logic are not violated.

Given that DTS exists for some applications, "free" power savings can be attained at no performance cost and no risk to timing safety by adjusting the operating voltage of the processor to exploit DTS while leaving design guardbands in place.

III. Quantifying DTS in Processors

Ultra-low-power embedded systems are a promising context for exploiting DTS, since embedded applications typically do not use all of the hardware features provided by a processor. Such applications may, therefore, not exercise the most timing critical logic in a processor [14], [11], [15]. Also, ultra-low-power processors are optimized to minimize area and power rather than maximize performance (e.g., many common microcontrollers have a small number of pipeline stages[3]), which typically results in relatively less balanced logic across pipeline stages or more delay variation across processor logic within a pipeline stage. This may increase available DTS, since in a design with larger delay variation, finite options for cell drive strength, threshold voltage, layout, etc. mean that not all paths will become timing-critical after design optimization.[4] Previous research has also observed that only a fraction of logic in an embedded design may be timing-critical [16].

In this section, we present measurement-based evidence of DTS in common ultra-low-power processors. We performed measurement experiments on four ultra-low-power processors to see whether DTS could be observed. We tested two processors (PIC24FJ64GA002 and MSP430F1610) by fixing the operating frequency and lowering the supply voltage to observe whether different applications exhibit different minimum safe operating voltages. For the other two processors (PIC16F88, and PIC18LF4550), we fixed the supply voltage and measured the maximum safe operating frequency for each application. Figure 3 shows the experimental setup used for the measurement experiments. The voltage of the processor is supplied by an external voltage supply, the frequency is regulated by an external function generator, and an ammeter is used to measure current.

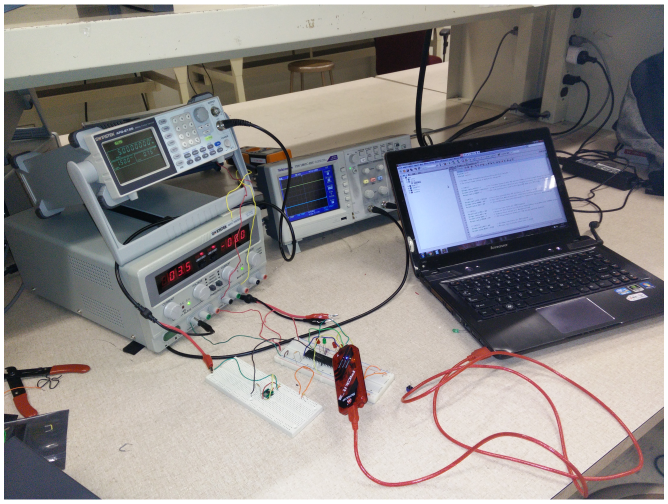


Fig. 3. Test setup for DTS measurement experiments.

For DTS testing on PIC24 and MSP430, frequency is held constant, voltage is lowered by increments of 0.01 V from the nominal voltage (2.00 V for PIC24, 3.30 V for MSP430), and the application under test is executed 1000 times to confirm correct operation. Table I reports the minimum voltage at which each application operates without errors, along with the power savings with respect to operation at the nominal voltage. The observed V_{min} varies by up to 120 mV for different applications on PIC24 and by up to 160 mV on MSP430, suggesting existence of significant DTS.[5]

TABLE I: Observed V_{min} on PIC24 and MSP430 for different sensor network benchmarks [17].

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

For DTS testing on PIC16 and PIC18 processors, we held voltage constant at each chip's nominal voltage and increased frequency beyond the maximum rated frequency (20 MHz for PIC16, 48 MHz for PIC18) in increments of 0.5 MHz. Table II reports the maximum

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

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

[5]Some fraction of the V_{min} differences across benchmarks in measured results could theoretically be due to input-dependent voltage and temperature variations, in spite of ultra-low currents. Results reported in Section VI isolate the impact of DTS alone, since they are captured assuming worst case variations and inputs.

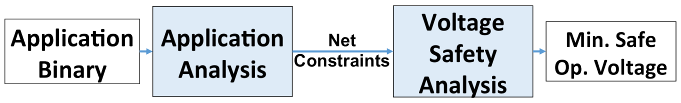


Fig. 4. We propose a tool that automatically identifies DTS by analyzing an application binary to determine parts of a design that cannot be exercised by the application and subsequently analyzing the timing safety of the constrained design at different voltages to determine the minimum safe operating voltage that exploits DTS for power reduction.

frequency at which each application operates without errors. Across benchmarks, we observed a difference in f_{max} of 4.0 MHz (20% of rated f_{max}) for PIC16, and 6.5 MHz (14% of rated f_{max}) for PIC18. This provides further evidence that significant DTS may exist for ultra-low-power processors.

In the next section, we present an automated methodology that identifies how much DTS is guaranteed to exist for a given processor and application under worst case conditions and determines the minimum operating voltage at which the application is guaranteed to execute safely at a given frequency.

TABLE II: Observed f_{max} on PIC16 and PIC18 for different sensor network benchmarks [17].

	$f_{max}(MHz)$	
Benchmark	PIC16	PIC18
binSearch	40.5	51.0
div	40.0	52.5
inSort	42.5	55.5
intAVG	41.5	53.5
intFilt	41.0	52.5
mult	42.0	55.5
rle	44.0	57.5
tHold	43.0	55.0
tea8	42.5	55.5

IV. Identifying and Exploiting DTS

To exploit DTS, we need a methodology that determines the minimum operating voltage at which an application is guaranteed to work correctly irrespective of the input and operating conditions. This automatically precludes all profiling or measurement-based approaches, since such approaches cannot guarantee that the minimum operating voltage determined during profiling is safe under all conditions; the voltage may not be safe when either the operating conditions change (e.g., temperature increases, the chip experiences aging-based degradation, the application is run on a different chip with a different amount of process variations, etc.) or the application is executed with a different input set. Instead, our approach for DTS identification and exploitation, illustrated in Figure 4, is based on analyzing an embedded software application to identify which parts of a design (e.g., registers, paths) cannot be exercised by the application *for any input* and subsequently determining the minimum operating voltage of the design such that parts of the design that can be exercised by the application are guaranteed to meet timing constraints *even under worst case operating conditions*. Parts of the design that cannot be exercised by the application are allowed to violate timing constraints. The approach is automated and determines the minimum safe operating voltage for an embedded application on a processor *without any programmer intervention*. Thus, an application designer need only provide an application binary; our tool automatically analyzes the binary and processor to determine the minimum voltage at which the processor can safely execute the application without reducing operating frequency at all.

As shown in Figure 4, the first stage of our DTS identification approach analyzes an application to determine the parts of a design that cannot be exercised by the application. This *application analysis* stage takes an application binary as input and evaluates the application's control flow graph (CFG) symbolically on the processor to determine which logic the application could possibly exercise (toggle) and which logic the application can never toggle. Analysis is input-independent, and information specifying which parts of the processor cannot be toggled by the application are passed to the next analysis stage in the form of design constraints. Our automated application analysis is described in detail in Section IV-A.

The second stage of our DTS identification approach takes as input the constraints (i.e., nets in the design that can never be toggled by the application) identified during application analysis and performs a constrained timing analysis on the processor design to determine the minimum safe operating voltage for the constrained design. Constrained timing analysis is performed at worst-case conditions (i.e., the slow process corner assuming worst-case process, voltage, temperature, and aging variations) to ensure that the selected voltage is safe under all possible operating conditions without reducing any design guardbands. The minimum safe operating voltage is chosen such that all parts of the processor that can be exercised by the application are guaranteed to meet timing constraints. Even though some processor logic may not meet timing constraints at the minimum safe voltage determined by voltage safety analysis, the application is still guaranteed to execute correctly since application analysis guarantees that this logic will not be toggled by the application. Thus, if our automated DTS identification approach identifies a more aggressive, safe operating voltage for an application, then operating at the new voltage results in free power savings, since the processor executes the application correctly without reducing frequency or performance at all. Section IV-B describes voltage safety analysis in detail.

A. Application Analysis

The goal of application analysis for DTS identification is to identify logic in the processor that an application is guaranteed to never exercise for any possible execution of the application. DTS is exposed for an application when un-exercised logic contains or contributes to the delay of timing-critical logic in the processor. DTS depends on two factors – a processor's functionalities (i.e., the architecture) and how those functionalities are used (or not used) by the processor's embedded software application. Application analysis performs symbolic evaluation of an application's CFG on the processor and observes which logic (specifically, which nets) in the processor cannot be toggled by the application.

Our automated approach for determining which nets in a processor design can be toggled by an application is based on the intuition that the embedded software application executing on a processor determines the possible states and state transitions that the processor can and cannot express. For example, an application that contains no multiply instructions will never exercise the logic in the processor's hardware multiplier. To fully explore which logic an application can exercise in a processor, our application analysis tool creates a CFG from the application binary and explores the possible paths through the CFG in breadth-first fashion. Each CFG path corresponds to a sequence of instructions, and CFG exploration involves performing gate-level simulation of the instruction sequences on the processor while monitoring the values of nets in the design. If a net toggles for any of the possible CFG paths, the net can be exercised by the application and will not be constrained. If a net maintains a constant value during exploration of the possible CFG paths, the net cannot be exercised by the application, and our tool generates a constraint specifying the constant value of the net for the application.

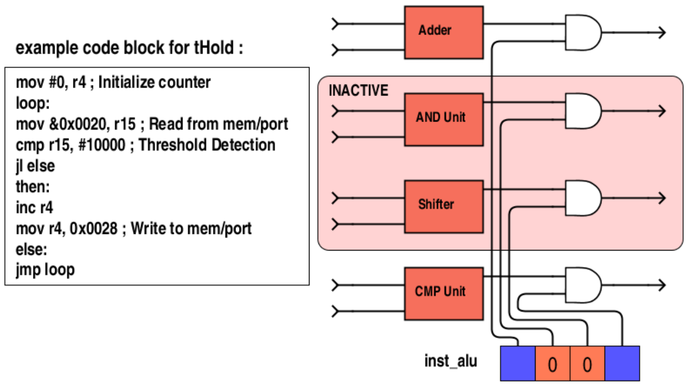
Because the logic exercised by an application can depend on the application's input data, we perform data-independent application analysis by injecting "don't care" (X) values into the processor logic whenever it reads an input value. This is equivalent to making worst case assumptions on input data. Any net that is assigned a value of X during CFG exploration cannot be constrained. In this way, the net constraints reported by application analysis *guarantee* that the constrained nets can *never* be toggled by the application for any possible input set. Algorithm 1 describes our automated approach for DTS identification through application analysis.

Algorithm 1 Application analysis for DTS identification

```
    Procedure Identify Constraints(app_binary, netlist)
 1. Add all nets to net_constraints[] // initially, constrain all nets
 2. Create CFG from app_binary and identify CFG paths $P_{CFG}$
 3. foreach path $p \in P_{CFG}$ do
 4.     Perform symbolic execution for p, using X for input values
 5.     if net n toggles or is assigned with X during symbolic execution then
 6.         Remove n from net_constraints[] // cannot constrain nets that can toggle
 7.     end if
 8. end for
 9. foreach net $n \in$ net_constraints[] do
10.     Record constrained value of n in net_constraints[n]
11. end for
12. return net_constraints[]
```

For an example of how application analysis can automatically identify un-exercised logic in a design that can be constrained to expose DTS, consider the `inst_alu` register in openMSP430. This 12-bit one-hot encoded register selects the function unit that will execute an instruction. A bit selecting a particular function unit will be set by an instruction that executes on the function unit. Not all applications utilize the entire instruction set, and a bit in `inst_alu` will not be toggled by an application that does not use the function unit selected by the bit. For example, none of the applications in our benchmark set use the `DADD` instruction (for BCD addition). Thus, the select bit corresponding to this instruction's function unit remains a constant zero during application analysis. As an example of different DTS for different applications, *rle* does not use right shift or left shift instructions, but *tea*8 does. Thus, application analysis reports a constraint for the shifter select bit in `inst_alu` for *rle* but not for *tea*8.

Figure 5 illustrates how application analysis can automatically identify constraints for un-exercised nets in `inst_alu` with a simplified example (i.e., a processor with only 4 operation types). As described above, a select bit in `inst_alu` only toggles during the execution of an embedded software application if the application contains an instruction that executes on the function unit selected by the bit. Exploring the CFG of the (*tHold*) application generates toggles in the adder and comparator select bits in `inst_alu` (colored blue in Figure 5), since *tHold* contains an `inc` instruction (which executes on the adder) and a `cmp` instruction. The code does not contain any `and` or `shift` instructions, however, so the corresponding select bits remain constant zero during application analysis of *tHold*. Applying these constraints propagates a controlling value to the select gates for the corresponding function units and eliminates the logic (labeled inactive) from consideration during timing analysis, potentially exposing DTS. Section A provides several



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

Fig. 5. DTS identification enabled by application analysis.

additional examples of constraints determined by application analysis for applications executing on openMSP430.

B. Voltage Safety Analysis

Voltage safety analysis takes as input the constraints identified during application analysis (Algorithm 1), applies them to the gate-level netlist, and performs timing analysis on the constrained design to determine the minimum voltage at which the design is guaranteed to operate safely for a given application. Voltage safety analysis is performed for worst case timing conditions such that the minimum operating voltage reported is guaranteed to be safe independent of PVT variations. Like application analysis, voltage safety analysis is fully automated and requires no effort on the part of the programmer. The combination of application analysis and voltage safety analysis forms an end-to-end automated tool that takes an application binary as input and reports the minimum operating voltage at which the processor can safely execute the application.

Voltage safety analysis is based on the observation that if part of a processor design cannot be exercised by the embedded software application, then it can be constrained to a constant value or ignored during design timing analysis to expose DTS and reveal a more aggressive operating voltage. During voltage safety analysis, net constraints reported by application analysis are applied to the gate-level netlist. Propagating these constants through the gate-level netlist can identify more constrained logic by propagating controlling values to logic gates. Any logic with a constant controlling value cannot toggle and thus can be ignored during timing analysis. A controlling value is defined for a gate as a value that, when assigned to an input pin of the gate, uniquely determines the output of the gate. For example, the controlling value of an AND/NAND gate is '0', because when any input to an AND/NAND gate is '0', the output is controlled to '0'/'1', regardless of the value of the other input. Similarly, the controlling value of an OR/NOR gate is '1'. An XOR gate does not have a controlling value.

Figure 6 describes the significance of propagating constraints for voltage safety analysis. If application analysis reveals that *FF2* cannot toggle for a particular embedded software application (e.g., if *FF2* is a register in the multiplier and the application contains no multiply instructions), then all paths terminating at *FF2* cannot toggle for the application (since a toggle on any path terminating at *FF2* implies a toggle of *FF2*). As another example, if application analysis reveals that *FF3* cannot toggle for an application and is constrained to a value of '0', then the path *FF1-G1-G2-G3-G4-FF2* also cannot toggle for the application, since an input to one of its gates (*G1*) is constrained to a controlling value. If the path in question is a critical path in the

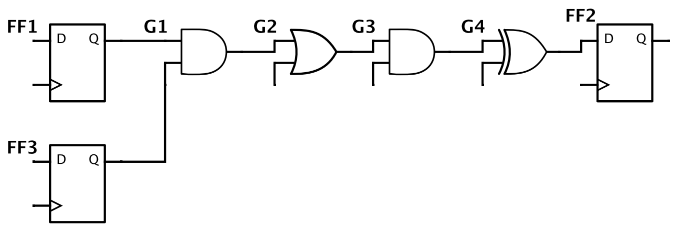


Fig. 6. Example circuit to illustrate the potential of design constraints to expose DTS.

design, then constraining the path may expose DTS.

All paths that pass through an un-toggled net or gate can be ignored during voltage safety analysis for an application. Such paths, by definition, are not toggled by the application, and the application will complete successfully even if these paths do not meet timing constraints. Voltage safety analysis ensures that all other paths in the design (the exercisable paths) meet timing constraints.

Once all possible constraints identified by application analysis have been applied to a design, voltage safety analysis checks whether all of the exercisable paths remaining in the design (e.g., the dynamic critical paths) meet timing constraints. The minimum safe operating voltage for the constrained design is determined by lowering the voltage in steps and performing constrained timing analysis at each step to find the lowest voltage at which all paths in the constrained design meet timing constraints (details in Section V). Algorithm 2 describes the voltage safety analysis stage of our automated DTS identification approach. Applying constraints to the netlist simply involves assigning constant values (the constraints identified by Algorithm 1) to nets in the design. The function *GetDynamicCriticalPath* uses STA to find the longest delay path in the constrained design and therefore has linear time complexity with the number of gates and nets in the circuit. On our system (system details in Section V), Algorithm 2 takes a maximum of 10 minutes and 10 seconds for OpenMSP430 (7218 gate design) to report the dynamic critical path at the 61 voltage levels spaced at 10 mV intervals between 1.00 V and 0.40 V.

Algorithm 2 Voltage safety analysis for DTS identification

```
Procedure Find $V_{min}(net\_constraints[], netlist)$
1.  Read netlist and initialize PrimeTime Tcl socket interface
2.  $V_{min}, V \leftarrow V_{nominal}$
3.  foreach constraint $c \in net\_constraints[]$ do
4.      Apply $c$ to netlist // using set_case_analysis
5.  end for
6.  $P_V \leftarrow$ GetDynamicCriticalPath()
7.  $S_V \leftarrow$ GetSlack($P_V$)
8.  while $S_V \geq 0$ do
9.      $V \leftarrow V - 0.01$
10.     $P_V \leftarrow$ GetDynamicCriticalPath() // report longest delay path at voltage V
11.     $S_V \leftarrow$ GetSlack($P_V$) // report DTS at voltage V
12.     if $S_V \geq 0$ then
13.         $V_{min} \leftarrow V$ // if design meets timing constraints, this voltage is safe
14.     end if
15. end while
16. return $V_{min}$
```

C. Enhancing DTS Benefits Through Dynamic Toggle Monitoring

If it is not possible to statically determine whether a net can toggle based on application analysis, then the net cannot be constrained during voltage safety analysis to expose DTS. However, it may be desirable to constrain some nets of this type in order to expose more DTS. For such scenarios, we propose microarchitectural support in the form of a simple circuit that detects a signal transition and sends a control signal to the voltage regulator to adapt the operating voltage to a safe level for the impending transition. When the monitored signal transitions back to its original value, the circuit stops asserting the control signal and the voltage regulator returns to the original aggressive voltage for the embedded software application.

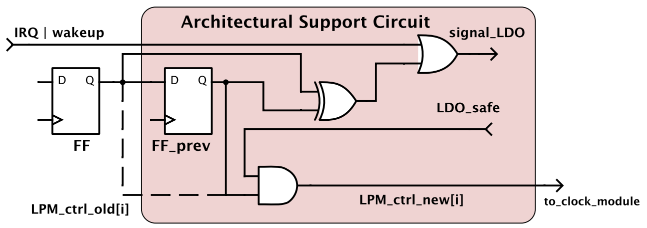


Fig. 7. Microarchitectural support for toggle detection and voltage adaptation – example for interrupt / low-power mode wakeup in openMSP430.

Figure 7 shows an example of the proposed microarchitectural support for interrupt / low-power mode wake-up detection and adaptation in openMSP430. The circuit adds an extra FF to create a 2-bit shift register with the monitored low-power mode register bit (*LPM_ctrl*). Different values between the FFs indicate a transition. An *IRQ* or *wakeup* signal or a transition of a low-power mode register bit sends a control signal (*signal_LDO*) that tells the voltage regulator to transition to a safe voltage (nominal) for the impending mode transition. Support circuitry delays the mode transition until an ack signal (rising edge of *LDO_safe*) from the voltage regulator indicates that the voltage has stabilized at the safe level for execution of the mode transition sequence. Once wake-up is complete, the voltage of the processor can be returned to the minimum safe operating voltage for the embedded software application. This happens when *signal_LDO* goes low after any *IRQ* or *wakeup* signal is reset (automatically by the processor) and the operating mode transition is complete (i.e., *FF* = *FF_prev*).

In most scenarios, there are no constraints on how quickly the supply voltage must be adjusted after detecting a toggle. The slew rate simply determines the performance overhead of stalling program execution on a toggle detection until the supply voltage adjustment is complete.[6] In case of some hard realtime systems, a slow voltage adjustment may cause missed hard deadlines, leading to correctness problems. The minimum slew rate for such systems will be determined by the realtime deadline and the latency jitter that the system was designed for. For many realtime systems, especially ones that interface with buses (e.g., CAN, FlexRay, or RealTimeEthernet), allowable jitter is in the ms to μs range. For such systems, even a slow off-chip voltage regulator will suffice, since the voltage adjustment required to exploit DTS would take around 10 μs at most with a slow regulator [18]. Finally, note that the microarchitectural support proposed in this section is a strictly optional mechanism provided to designers for exploiting additional DTS in their designs.

V. Experimental Methodology

We verify the existence of DTS and evaluate the power benefits available through DTS exploitation for a silicon-proven ultra-low-power processor – openMSP430 [8]. The processor is synthesized, placed, and routed with TSMC 65GP library (65nm), using Synopsys Design Compiler [19] and Cadence EDI System [20]. The processor

[6]Since exceptions, interrupts, and power mode changes are typically rare, dynamic toggle detection for these events would have insignificant impact on performance in most scenarios.

is optimized at the highest optimization level, including aggressive cell sizing and multiple threshold voltages, to minimize slack as much as possible. To analyze design timing and power at different voltages, Cadence Library Characterizer was used to generate libraries at each voltage (V_{dd}) between 1.0V and 0.5V at 0.01V intervals. The processor was implemented at 100 MHz. Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using Synopsys VCS [21]. We show results for all benchmarks from [17] 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 [17]. In addition to evaluating a bare-metal environment common in ultra-low-power embedded systems, we also evaluate DTS for our applications running on the processor with an operating system (FreeRTOS [22]). Timing and power analyses are performed with Synopsys PrimeTime [23]. Experiments were performed on a server housing two Intel Xeon E5-2640 processors (8-cores each, 2GHz operating frequency, and 64GB RAM). Our application and voltage safety analysis tool is implemented in C++.

TABLE III: BENCHMARKS

Embedded Sensor Benchmarks [17]
mult, tea8, binSearch, rle, intAVG, inSort, tHold, div, intFilt
EEMBC Embedded Benchmarks
AutoCorr, ConvEnc, FFT, Viterbi
Operating System
FreeRTOS

VI. Results

To validate the software implementation of our toolflow, we ran gate-level simulations for the applications in our benchmark set and confirmed that the constrained paths reported by application analysis indeed did not toggle during execution.[7]

Table IV presents the power reduction for openMSP430 afforded by DTS identification based on automated application analysis and voltage safety analysis. The table shows the amount of DTS exposed by our tool as a percentage of the clock period, the minimum safe operating voltage reported, and the power savings afforded for each application. The voltage reduction allowed from exploiting DTS is non-speculative and requires no reduction in operating frequency, so reported benefits are essentially "free" power savings. Our baseline for the results is the processor operating at nominal frequency and voltage (100MHz and 1V for openMSP430).

The results in Table IV show that different applications can expose different amounts of DTS, resulting in different minimum safe operating voltages. This is because different applications exercise different processor features, resulting in a different set of logic constraints. For example, relatively less DTS is available for *AutoCorr*, *FFT*, *intFilt*, and *mult* because these applications use the hardware multiplier, which is one of the most timing-critical modules in the processor (Table V lists the most timing critical registers in the design – most registers contain multiple endpoints). Among these benchmarks *AutoCorr* and *FFT* have the least DTS, since they can exercise the critical multiplier paths terminating at the peripheral data capture register (exercised when multiplier operands are read from registers

[7]For our benchmarks, gate-level simulation to completion for MSP430 took an average of 16.7 s (minimum of 12.1 s for a 70119 cycle application and maximum of 40.8 s for a 173412 cycle application).

TABLE IV: POWER SAVINGS FROM EXPLOITING DTS

	Benchmark	DTS (%)	Voltage (V)	Power Savings (%)
Embedded Sensors	binSearch	32.01	0.86	28.38
	div	31.42	0.86	28.42
	inSort	31.79	0.86	28.40
	intAVG	31.73	0.86	28.36
	intFilt	21.64	0.90	21.10
	mult	12.34	0.94	13.54
	rle	31.76	0.86	28.34
	tHold	32.13	0.86	28.37
	tea8	31.42	0.86	28.36
EEMBC	AutoCorr	10.04	0.95	11.15
	ConvEnc	31.73	0.86	28.39
	FFT	10.04	0.95	11.14
	Viterbi	31.43	0.86	28.50

TABLE V: TIMING-CRITICAL REGISTERS AND THEIR WORST SLACK VALUES (OPENMSP430)

Endpoint / Functionality	**Worst Slack** (%)
Data synchronization for DCO wakeup	0.80
Low frequency oscillator disable	1.74
Digitally Controlled Oscillator (DCO) disable	1.86
Clock gating / enable master clock	2.28
Peripheral data capture	9.94
Multiplier overflow	17.62
Program memory data backup	21.87
Upper bits of multiplier	22.55

and the result overflows). *mult* has slightly more DTS, since its dynamic critical path terminates at the multiplier overflow register (exercised when input operands are read from memory and the result overflows). *intFilt* has even more slack, since the coefficients of the filter in *intFilt* are all less than one and cannot cause overflow.

Our tool reports similar DTS for several benchmarks. This is because several embedded benchmarks exercise similar hardware resources, especially in an in-order processor. All studied applications have some DTS, since the static critical paths of the design go through clock and power management circuitry (Table V), and none of our benchmarks change the clock source or the processor power state. Thus, they do not exercise the static critical paths and as a result, they have dynamic timing slack. [8] Over the range of applications, average power savings from exploiting DTS are 25%.

DTS can also be exploited to increase frequency without increasing the voltage. The first two columns in Table VI show the maximum increase in performance possible from exploiting DTS, along with the corresponding power increase. It is also possible to exploit DTS to maximize performance for the same power budget by reducing voltage and increasing frequency in tandem. The "Iso-power" columns in Table VI show the maximum performance increase possible from exploiting DTS while maintaining a constant power budget, along with the corresponding operating voltage.

Note that our voltage safety analysis is performed at the worst case (slow) corner, so the operating voltage determined by our analysis is guaranteed to be safe even when the design is affected by worst case variations. This comes at the expense of power benefits. We calculated the average power cost of performing analysis at worst case rather than typical case as 23%. In Section VI-A we discuss how better-than-worst-case design techniques can be combined with DTS analysis to mitigate this cost.

[8]Applications show even more difference in their minimum voltages in our measured results in Section III.

TABLE VI: Performance improvement from exploiting DTS

	Benchmark	Maximum Perf. Inc. (%)	Power Inc. (%)	Iso-power Perf. Inc. (%)	Iso-power Voltage (V)
Embedded Sensors	binSearch	47	37	22	0.93
	div	45	35	22	0.93
	inSort	46	36	22	0.93
	intAVG	46	36	22	0.93
	intFilt	27	21	15	0.96
	mult	14	11	6	0.98
	rle	46	36	21	0.92
	tHold	47	37	22	0.92
	tea8	45	35	22	0.93
EEMBC	AutoCorr	11	8	6	0.98
	ConvEnc	46	27	22	0.93
	FFT	11	8	6	0.98
	Viterbi	45	36	22	0.93

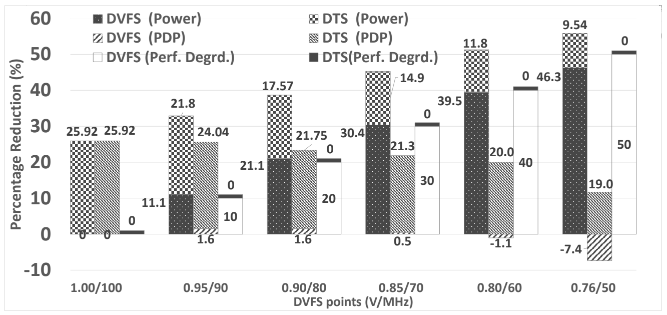


Fig. 8. DVFS reduces frequency along with voltage and may lead to performance degradation. DTS enables voltage reduction without any reduction of frequency. Furthermore, the benefits of DTS are orthogonal to those of DVFS and can be extracted in addition to any benefits produced by DVFS.

Finally, to ensure safety our analysis makes worst-case assumptions about input data (Section IV-A). Benefits may be higher if input data are known. For example, for a benchmark like *mult*, the slack difference can be significant across inputs, since many multiplier paths that are near-critical are only exercised for particular inputs (e.g., inputs that cause overflow). In fact, while a guaranteed safe voltage of 0.94 V was reported by our input-independent analysis (Table IV), the benchmark is able to operate safely at 0.85 V when small input values that do not cause overflow are used.

A. Comparison with Related Work

DVFS:

In this work, we exploit DTS for power reduction by reducing voltage without reducing frequency, such that all exercised parts of a processor design meet timing constraints. Related work on DVFS [24], [25], [26] also reduces power by reducing voltage; however, DVFS reduces frequency along with voltage to ensure timing safety. Figure 8 compares power and energy reduction achieved by DVFS and DTS at different DVFS operating points (V/f). The “Power” series in Figure 8 shows power reduction for DVFS along with additional power reduction achieved by exploiting DTS at each operating point. DTS is orthogonal to DVFS and as such, DTS can be exploited for additional power savings at any DVFS operating point, even at the nominal operating point, where DVFS is not exploited to reduce power. Furthermore, while DVFS may lead to significant performance reduction, especially when performance is strongly correlated with frequency (e.g., in a system with embedded memories, like openMSP430), exploiting DTS at any operating point introduces no additional performance degradation, since DTS allows voltage reduction without any frequency reduction. Results for power-delay product (PDP) reduction show that DVFS can lead to an increase in energy (negative PDP reduction) at some operating points while DTS always reduces energy.

Better-than-worst-case Design:

Our DTS identification methodology performs analysis at the worst-case design corner, leaving unexploited benefits at better-than-worst-case (BTWC) operating conditions. Below, we describe how our DTS identification approach can be combined with BTWC design techniques to reclaim benefits of guardband reduction while safely exploiting DTS. We also compare our approach for exploiting DTS against two popular BTWC design techniques – critical path monitors (CPMs) [27] and Razor [6].

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

The timing slack in guardbands under BTWC conditions (exploited by CPMs) is orthogonal to DTS (timing slack between un-exercised static critical paths and exercised dynamic critical paths). As such, DTS exploitation techniques can be used synergistically with CPMs for additional power reduction by using CPMs to track dynamic critical path delay rather than static critical path delay. We refer to CPMs that track dynamic critical path delay as dynamic critical path monitors (DCPMs) as opposed to conventional static critical path monitors (SCPMs). Since our DTS identification techniques identify the dynamic critical paths exercised by an application, tuning CPMs to track dynamic critical path delay is feasible using tunable CPMs [28], [29].

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

To evaluate Razor, we first identify flip-flops (FFs) that need to be replaced with Razor FFs by selecting the minimum safe operating voltage for the processor under typical case operating conditions and identifying all the FFs that can violate timing constraints at this voltage under worst case operating conditions. After replacing these FFs with Razor FFs containing an extra (shadow) latch, clock buffer,

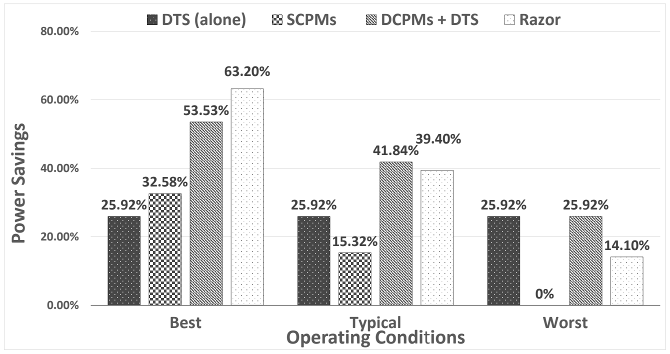


Fig. 9. Comparison of power savings for DTS, Razor, and CPMs under different operating conditions.

XOR gate for error detection, and MUX for error correction, an "OR" network was added to combine the error signals to be sent to the voltage regulator, and hold time constraints were placed on the Razor FFs during layout of the synthesized netlist to generate the placed and routed netlist. Implementing Razor in this fashion resulted in an area overhead of 14% for openMSP430. Note, however, that this is an optimistic evaluation of Razor, as the Razor overheads for metastability detectors, error correction (dynamic performance and power overheads), clock gating, error rate measurement, and voltage control logic were not considered. Also, the design was not able to meet the hold time constraint for all Razor FFs (one of several difficult challenges for Razor designs [6]). Although we did not account any error correction overheads, we evaluated the Razor-based design at a reduced voltage corresponding to a 1% error rate for each benchmark.

Figure 9 compares power reduction achieved by DTS exploitation, SCPMs, DCPMs+DTS, and Razor under different operating conditions (worst, typical, best). SCPMs achieve significant power reduction at BTWC operating points (typical, best) but no reduction under worst case conditions. DTS, however, can be exploited for significant power savings (25%) even in worst case conditions. Exploiting DTS synergistically with CPMs (DCPMs+DTS) achieves significant additional benefits over SCPMs at BTWC operating points.

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

B. Generality and Limitations

The above results show that the proposed approach for identifying and exploiting DTS is very effective at improving the energy efficiency of ultra-low-power embedded systems. In this section, we discuss the efficacy of the proposed approach in other contexts.

Complex processors: DTS requires that the subset of exercised logic for a given application is not timing-critical. Unlike ultra-low-power processors, which are optimized to minimize power and area, processors that are optimized for high performance may exhibit more balanced logic delays (*slack wall [9], [10]*) and consequently less DTS. At the same time, prior work shows that only 7 - 15% of flip-flops in the Alpha processors in [12], [13], [30] were near-critical. UltraSparc T2 was reported to have similar behavior [10]. It may be possible to exploit DTS for such designs.

More complex processors also contain performance-enhan-cing features such as large caches, prediction or speculation mechanisms, and out-of-order execution, that introduce non-determinism into the instruction stream. Our application analysis is capable of handling this added non-determinism at the expense of analysis tool runtime. For example, by injecting an X as the result of a tag check, both the cache hit and miss paths will be explored in the memory hierarchy. Similarly, since our application analysis tool already explores taken and not-taken paths for input-dependent branches, it can be easily adapted to handle branch prediction. Our approach for input-independent CFG exploration is easily modifiable to perform input-independent exploration of the data flow graph (DFG), and thus can be made to analyze out-of-order execution. Finally, even in the worst case where DTS cannot be exploited for a complex processor, we note that DTS benefits are high for ultra-low-power embedded processors (see Sections III and VI). These processors are already ubiquitous and are also expected to power a large number of emerging applications [1], [2], [3], [4], [5].

Complex Applications: Our automated application analysis approach employs techniques similar to symbolic execution [31], [32], where X symbols are propagated for input values. One limitation of the general symbolic execution approach is a potential explosion of possible CFG paths as CFG complexity increases. This issue is ameliorated in the context of simple in-order processors (e.g., the ultra-low-power processors studied in this paper) because 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 the scalability of symbolic execution [33].

Also, complex applications may have more phases with distinct behavior and more complex CFGs. For such applications, per-phase application analysis may expose more DTS. Once phases have been identified, our application analysis tool can easily identify a unique minimum safe voltage corresponding to each phase. Phase adaptation can be achieved by instrumenting the binary to change the voltage prior to execution of each phase.

System Code: Our evaluations in Section VI 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 [34], [35][9], use of an OS is common in several embedded application domains as well as in more complex systems. Thus, we also evaluated DTS for our applications running on the processor with an OS (FreeRTOS [22]). System code was analyzed in addition to application code to identify available DTS.

Application analysis of system code for FreeRTOS reveals that many nets are not exercised by the OS, including the entire hardware multiplier. For the OS running with applications that do not use the multiplier, average power savings from exploiting DTS are 21.1%. For applications that use the multiplier, power savings are 11.2% with the OS.

In several settings, it may not be possible to analyze the system

[9]Many embedded processors provide toolchains for bare-metal development [36], [37].

code completely. To guarantee safety when exploiting DTS in such settings, any un-analyzed code must be run at nominal voltage. For example, this is a simple alternative to application analysis for handling startup code, bootloader, etc. that runs only briefly. Voltage can be reduced to exploit DTS after startup when the system enters user mode.

Multi-programming and Multi-threading: Multi-program-ming and multi-threading present a challenge for DTS analysis, since they can introduce non-determinism in the instruction stream executed by a processor. In a multi-program environment, different applications (including system code) may expose different amounts of DTS. In such a scenario, the metadata for a binary can incorporate the minimum safe voltage for the application reported by voltage safety analysis, and the processor can use dynamic voltage scaling at context switch time to adjust to the minimum safe voltage for each application or system code. While an application is resident, the value corresponding to the minimum safe voltage can be stored as part of the application's context. Entry into system code, which triggers voltage scaling, can be initiated by a timer interrupt or when the running process yields control to the OS through a system call. Voltage scaling in response to a timer interrupt is easily managed by the architectural support described in Section IV-C, and yielding control to the OS can be handled by performing voltage scaling in software as the first action of a system call.

DTS benefits may also be possible for fine-grained concurrent execution (e.g., block multithreading, SMT, etc.). For fine-grained execution, the minimum voltage of the processor is determined as the maximum of the minimum voltages reported by our analysis tool for the different threads. However, since it may not be possible to determine all possible interleavings of instructions between the threads, a minor adaptation to Algorithm 1 is needed to determine the safe minimum voltage for a thread that is agnostic to other threads' behavior. Any state that is not maintained as part of a thread's context is now assumed to have a value of X when symbolic execution is performed for an instruction belonging to the thread. All state that is part of the thread's context will be maintained, and thus, need not be set to X. This leads to a safe minimum voltage for the thread irrespective of the behavior of the other threads.

Dynamic Linking and Self-Modifying Code: For systems that support dynamic linking[10], our methodology can be used to identify the minimum safe voltages for the caller application, the called library function, and the OS code used for performing dynamic linking. The minimum safe voltage of the processor is the maximum of the three minimum voltages. Similarly, for self-modifying code, our methodology can be used to identify the minimum safe voltage for each code version. The maximum of these voltages will be chosen as the safe operating point for the processor.

VII. Conclusion and Future Work

In this paper, we present a novel opportunity for power savings in ultra-low-power processors – dynamic timing slack. DTS is observed because embedded applications do not always use all the features of a general purpose processor. As such, there may be timing slack between unused *static* critical paths of a processor and the *dynamic* critical paths that are exercised by an application. We show that DTS exists for a common ultra-low-power processor and propose a fully automated technique that is non-speculative and requires no programmer intervention for identifying and exploiting DTS for power savings. We demonstrate that exploiting DTS in an ultra-low-power processor results in power savings of up to 32%, 25% on average, without any performance degradation.

Future work includes:

• Design- and architecture-level optimizations that increase DTS by increasing timing slack on dynamic critical paths
• Compiler- and algorithm-level optimizations that increase DTS by eliminating activity on the dynamic critical paths exercised by a workload
• DTS-aware scheduling policies in multi-programmed and multi-threaded settings
• Exploiting DTS at a finer granularity (e.g., per-phase DTS adaptation)
• Leveraging symbolic simulation to enable other hardware optimization and analysis techniques

Acknowledgments

The authors would like to thank Jingjing Zhou, Taoran Wang, and Himanshu Shekhar Sahoo, who performed testing of ULP processors for Section III. This work was partially supported by NSF, SRC/GRC, and CFAR, within STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Appendix

Below, we present several examples that illustrate how DTS can be exposed in openMSP430 for various applications.

<u>*Execute*</u>: The execute stage contains the general purpose register (GPR) file and the function units. For a GPR to toggle, some instruction must use the GPR as a destination register. For a function unit to toggle, a binary must contain an instruction that executes on the function unit. If a certain GPR or function unit is not used at all by an application, application analysis reports a constraint for the register or opcode select bit. Figure 5 presents an example.

<u>*Fetch*</u>: The fetch stage in the processor frontend contains two key registers – the Program Counter (PC) and the Instruction Register (IR). The IR holds fetched instructions. Certain IR bits might not be toggled by an application if the set of instructions used by the application is limited.

While it should be possible to constrain the PC based on the range of addresses defined by the code area that a binary can access and the valid interrupt vectors for the application, we choose not to constrain the PC to ensure correct operation in case an error causes the processor to execute from an illegal address.

<u>*Decode*</u>: Constraints are reported for decode registers when some of the legal values of the decode registers are unused by an application. Since these register fields control logic in other parts of the pipeline, constraints here may propagate controlling values to other logic, resulting in additional DTS.

<u>*Memory*</u>: Since memory values are input-dependent, logic in the memory stage generally cannot be constrained, unless an application contains no load, store, or move instructions.

<u>*Write Back*</u>: Like in the case of GPRs, writeback to a register requires that the register is the destination register for some instruction. The select bits corresponding to registers that are never written are never toggled by a captive application and can be constrained.

Peripherals, Special Function Registers, Clock and Reset Management: These structures toggle only when a specific instruction sequence is executed. For example, the multiplier in openMSP430 is written to with a `mov` instruction to address `0x132` or `0x138`. If neither `mov` instruction appears in an application's binary, then the

[10]Very few embedded systems support dynamic linking [38], [39] (space constraints) or self-modifying code (needs RAM program memory instead of ROM – RAM is more expensive than ROM and also more susceptible to EM interference).

multiplier is not utilized at all and can be constrained to expose DTS. Other functionalities mentioned here have similar behavior.

Other Modes of Operation: Debug and scan modes are initiated by particular pins in the processor (i.e., mode entry or exit is controlled by setting or resetting the pin's value). These pins can typically be constrained to expose DTS, since scan and debug modes are typically only used during design testing and validation and not during system deployment.

When the processor exits a low-power mode, the corresponding state register toggles in response to either an external or timer-generated interrupt. While this condition (interrupt) cannot be determined statically, it is easy to detect by monitoring the interrupt request (IRQ) and wakeup pins in the processor (Section IV-C).

References

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

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

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

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

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

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

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

[8] O Girard. Openmsp430 project. *available at opencores.org*, 2013.

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

[10] A.B. Kahng, Seokhyeong Kang, R. Kumar, and J. Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 825–831, Jan 2010.

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

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

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

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

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

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

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

[18] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134. IEEE, 2008.

[19] Synopsys. *Design Compiler User Guide*.

[20] Cadence. *Encounter Digital Implementation User Guide*.

[21] Synopsys. *VCS/VCSi User Guide*.

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

[23] Synopsys. *PrimeTime User Guide*.

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

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

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

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

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

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

[30] S. Das, D. Roberts, Seokwoo Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. A self-tuning DVS processor using delay-error detection and correction. *Solid-State Circuits, IEEE Journal of*, 41(4):792–804, April 2006.

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

[32] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

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

[34] Hacking Pacemakers. http://spectrum.ieee.org/podcast/biomedical/devices/hacking-pacemakers/.

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

[36] StarterWare. http://processors.wiki.ti.com/index.php/StarterWare.

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

[38] Embedded systems. https://en.wikibooks.org/wiki/Embedded_Systems, 2015.

[39] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, November 2005.