Bespoke Processors for Applications with Ultra-low Area and Power Constraints

Hari Cherupalli
University of Minnesota
cheru007@umn.edu

Henry Duwe
University of Illinois
duweiii2@illinois.edu

Weidong Ye
University of Illinois
wye5@illinois.edu

Rakesh Kumar
University of Illinois
rakeshk@illinois.edu

John Sartori
University of Minnesota
jsartori@umn.edu

ABSTRACT

A large number of emerging applications such as implantables, wearables, printed electronics, and IoT have ultra-low area and power constraints. These applications rely on ultra-low-power general purpose microcontrollers and microprocessors, making them the most abundant type of processor 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 paper, 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%.

CCS CONCEPTS

• Computer systems organization → Special purpose systems; **Embedded systems**; **• Hardware → Application specific processors**;

KEYWORDS

ultra-low-power processors, application-specific processors, bespoke processors, hardware-software co-analysis, Internet of Things

ACM Reference format:
Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Bespoke Processors for Applications with Ultra-low Area and Power Constraints. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages. https://doi.org/10.1145/3079856.3080247

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080247

1 INTRODUCTION

A large class of emerging applications is characterized by severe area and power constraints. For example, wearables [46, 51] and implantables [25, 50] are extremely area- and power-constrained. Several IoT applications, such as stick-on electronic labels [48], RFIDs [54], and sensors [30, 72], are also extremely area- and power-constrained. Area constraints are expected to be severe also for printed plastic [49] and organic [41] 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 [5, 23, 53], in spite of their well-known inefficiency compared to ASIC and FPGA-based solutions [31]. 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 2). Also, these unused gates continue to consume power, resulting in significant power inefficiency. While adaptive power management techniques (e.g., power gating [58, 60]) 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 7). 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.

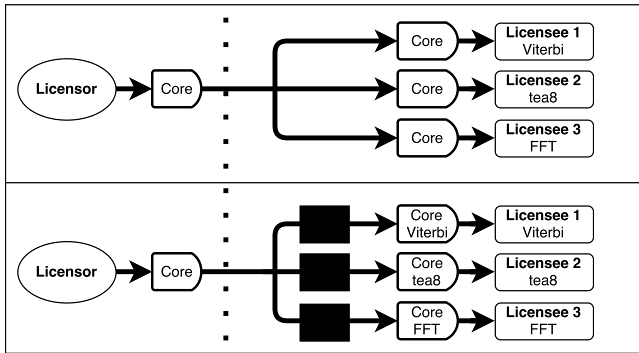


Figure 1: General purpose processors are overdesigned for a specific application (top). A bespoke processor design methodology allows a microprocessor IP licensor or licensee to target different applications efficiently without additional software or hardware development cost (bottom).

Also, since no logic used by the application is eliminated, area and power benefits come at no performance cost. The resulting bespoke processor does not require programmer intervention or hardware support either, since the software application can still run, unmodified, on the bespoke processor.

In this paper, 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 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 (Section 3.5).

This paper 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.

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 [5, 23, 34, 53]. 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. 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 [1]. 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 2 illustrates this point, showing the fraction of gates in an openMSP430 [27] processor that are not toggled when a variety of applications (Table 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 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 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

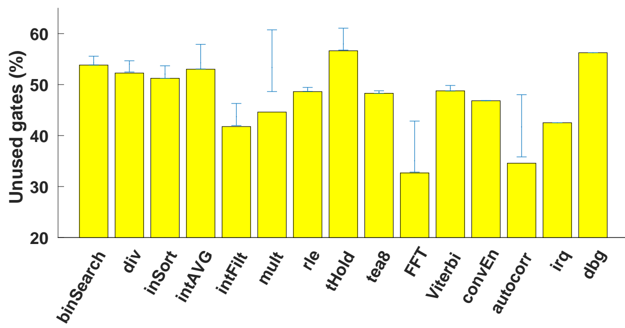
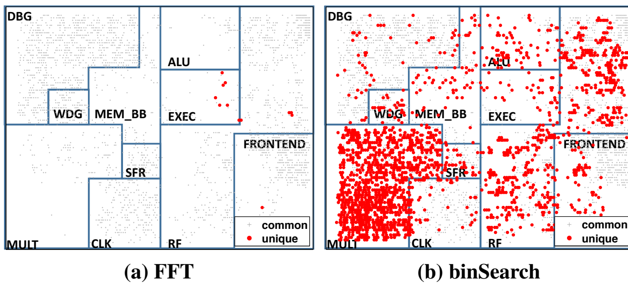


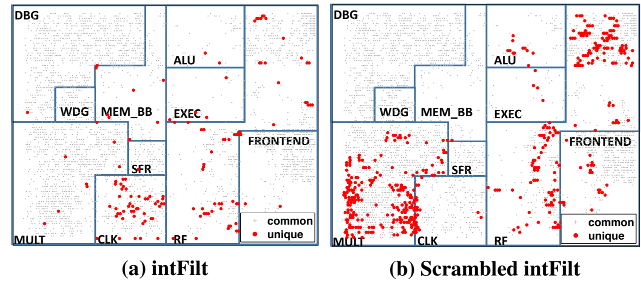
Figure 2: A significant fraction of gates in an openMSP430 processor are not toggled when an application executes on the processor. Each bar represents gates not toggled by any input for an application; the interval shows the range of unexercised gates for different inputs.



(a) FFT **(b) binSearch**

Figure 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.

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 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 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 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.



(a) intFilt **(b) Scrambled intFilt**

Figure 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.

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). However, 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.

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 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

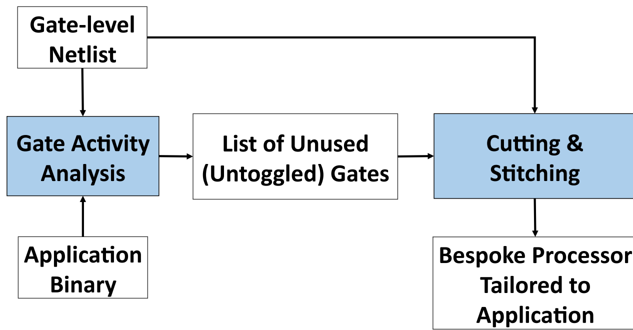


Figure 5: Our technique performs input-independent gate activity analysis to determine which gates of a processor cannot be toggled in any execution of the application. These gates are then cut from the design to form a custom, bespoke processor with reduced area and power.

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 the cut connections between gates to maintain functional equivalence to the original design for the target application.

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 have implemented an analysis technique based on symbolic simulation [7], that is able to characterize the gate-level activity of a processor executing an application for all possible inputs with a single gate-level simulation. During this simulation, inputs are represented as unknown logic values (Xs), which are treated as both 1s and 0s when recording possible toggled gates.

Symbolic simulation has been applied in circuits for logic and timing verification, as well as sequential test generation [8, 24, 37, 42, 44]. More recently, it has been applied to determine application-specific V_{min} [18]. Symbolic simulation has also been applied for software verification [74]. However, to the best of our knowledge, no existing technique has applied symbolic simulation to create bespoke processors tailored to the requirements of an application.

Algorithm 1 describes input-independent gate activity analysis. Initially, the values of all memory cells and gates are set to Xs. The application binary is loaded into program memory, providing the values that effectively constrain which gates can be toggled during execution. During simulation, our simulator sets all inputs to Xs, which propagate through the gate-level netlist during simulation.[1] After each cycle is simulated, the toggled gates are removed from the list of unexercisable gates. Gates where an X propagated are considered as toggled, since some input assignment could cause the gates to toggle. If an X propagates to the PC, indicating input-dependent control flow, our simulator branches the execution tree and simulates execution for all possible branch paths, following a depth-first ordering of the control flow graph. Since this naive simulation approach does not scale well for complex or infinite control structures which result in a large number of branches to explore, we employ a conservative approximation that allows our analysis to scale for arbitrarily-complex control structures while conservatively maintaining correctness in identifying exercisable gates. Our approximation works by tracking the most conservative gate-level state that has been observed for each PC-changing instruction (e.g., conditional branch). The most conservative state is the one where the most variables are assumed to be unknown (X). When a branch is re-encountered while simulating on a control flow path, simulation down that path can be terminated if the symbolic state being simulated is a substate of the most conservative state previously observed at the branch (i.e., the states match or the more conservative state has Xs in all differing variables), since the state (or a more conservative version) has already been explored. If the simulated state is not a substate of the most conservative observed state, the two states are merged to create a new conservative symbolic state by replacing differing state variables with Xs, and simulation continues from the conservative state. This conservative approximation technique allows gate activity analysis to complete in a small number of passes through the application code, even for applications with an exponentially-large or infinite number of execution paths.[2]

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.

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 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

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

[2] Some complex applications and processors might still require heuristics for exploration of a large number of execution paths [10, 32]; however, our approach is adequate for ULP systems, representative of an increasing number of future applications which tend to have simple processors and applications [36, 53]. For example, complete analysis of our most complex benchmark takes 3 hours.

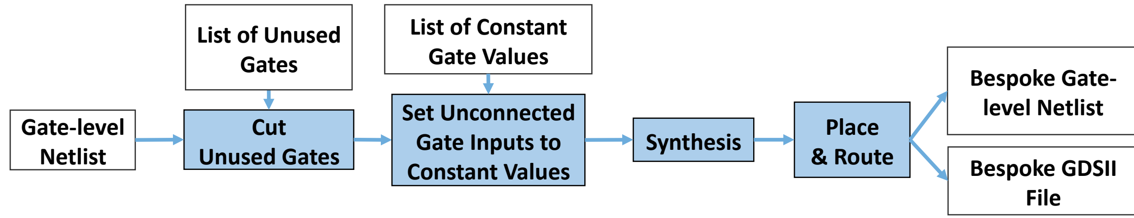


Figure 6: Tool flow for cutting and stitching.

Algorithm 1 Input-independent Gate Activity Analysis

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

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.

3.3 Illustrative Example

This section illustrates how bespoke processor design tailors a processor design to a particular application, as described in Sections 3.1 and 3.2. Figure 7 illustrates the bespoke design process. The left part of Figure 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.

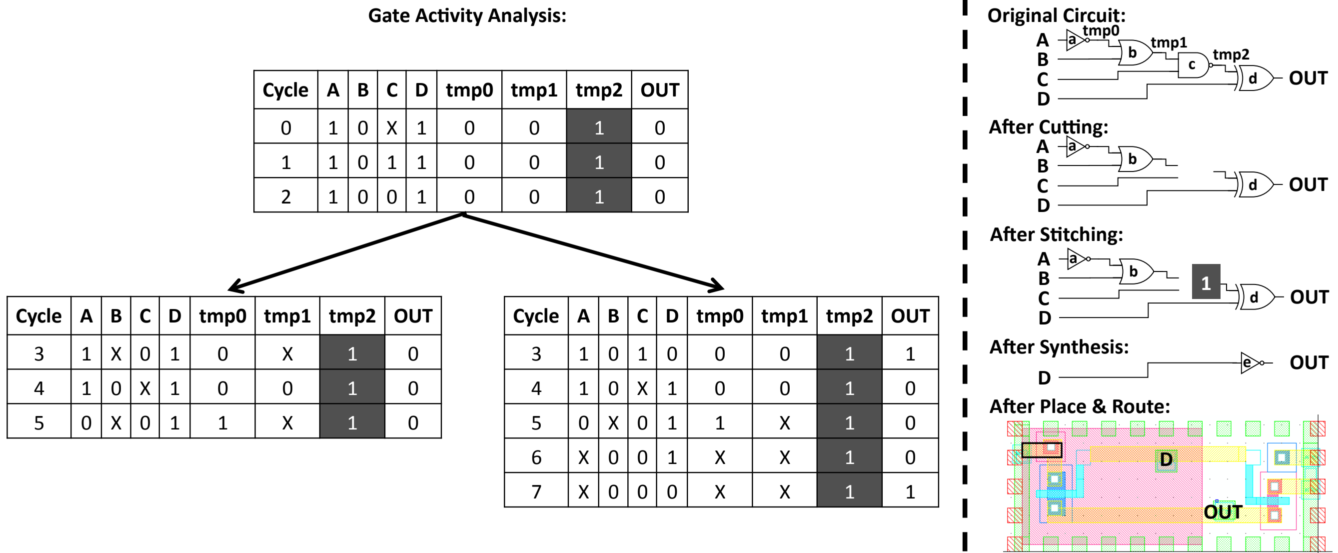
Once gate activity analysis has generated a list of cuttable gates and their constant values, cutting and stitching begins. Since gate c was marked for cutting, it is removed from the netlist, leaving the input to its fanout (d) unconnected. During stitching, d's floating input is connected to c's known 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.

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

Proof: The first step in creating $\mathcal{B}_{\mathcal{A}}$ – input-independent gate activity analysis (Section 3.1) – identifies the subset $\mathcal{E}$ of all gates in



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

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

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

Figure 7: An example of gate activity analysis and cutting and stitching.

the processor that can possibly be exercised by $\mathcal{A}$, for all possible inputs. The analysis also identifies the constant output values for all gates $\mathcal{U}$ that can never be exercised by $\mathcal{A}$. It follows that $\mathcal{E} \cap \mathcal{U} = \emptyset$ and $\mathcal{E} \cup \mathcal{U} = \mathcal{G}$, where $\mathcal{G}$ is the set of all gates in $\mathcal{P}$. Cutting and stitching (Section 3.2) removes all gates in the set $\mathcal{U}$ and ties their output nets to their known constant values, such that the functionality of all gates in $\mathcal{U}$ is maintained in $\mathcal{B}_{\mathcal{A}}$. All gates in $\mathcal{E}$ remain in the bespoke design, so all gates in $\mathcal{E}$ have the same functionality and produce the same outputs in $\mathcal{B}_{\mathcal{A}}$ and $\mathcal{P}$. Since $\mathcal{E} \cup \mathcal{U} = \mathcal{G}$, it follows that $\mathcal{B}_{\mathcal{A}}$ is functionally equivalent to $\mathcal{P}$ for $\mathcal{A}$ and produces the same output as $\mathcal{P}$ for all possible inputs to $\mathcal{A}$. ■

We also verified correctness through input-independent gate activity analysis and input-based simulations on both the original and bespoke processor for every application (Section 5.1), confirming that outputs of both processors were the same in each case.[3]

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 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 5) is simply expanded to support multiple target applications, as shown in Figure 8. Given a known set of binaries that need to be supported, gate activity analysis

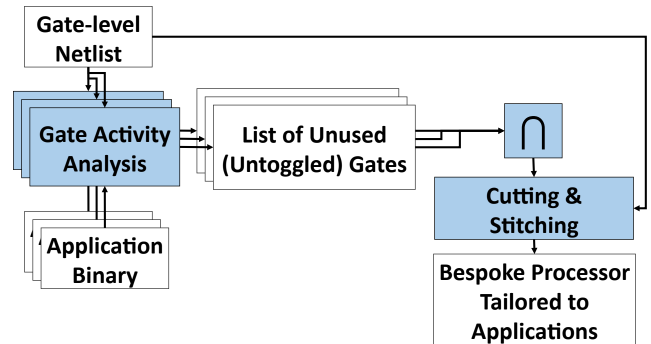


Figure 8: To support multiple programs, our bespoke design technique performs input-independent gate activity analysis on each program. Cutting and stitching is performed using the intersection of the untoggled gates lists from all supported programs.

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 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 invariably 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

[3] Industrial equivalence checking tools (e.g., Formality [63]) 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.

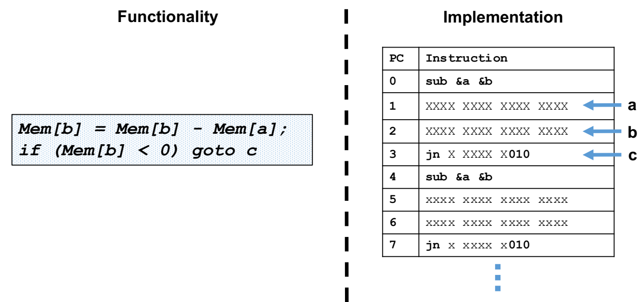


Figure 9: Functionality and MSP430 implementation of Turing-complete subneg pseudo-instruction.

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 [38], 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 subneg, an example Turing-complete instruction [26]. Figure 9 shows the functionality and code implementation of a subneg pseudo-instruction created for MSP430. Since the memory operand addresses and the branch target are assumed to be unknown values (Xs), a binary that characterizes the behavior of 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.

4 METHODOLOGY

4.1 Simulation Infrastructure and Benchmarks

We verify our technique on a silicon-proven processor – openMSP430 [27], an open-source version of one of the most popular ULP processors [6, 70]. 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 [62] and Cadence EDI System [11]. Gate-level simulations are performed by running full benchmark applications on the placed and routed processor using a custom gate-level simulator that efficiently traverses the control flow graph of an application and captures input-independent activity profiles (Section 3.1). Table 1 lists our benchmark applications. We show results for all benchmarks from [73] and all EEMBC benchmarks [22] that fit in the program memory of the processor. We also added unit test benchmarks [27] 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 [17, 56, 61, 66], we also evaluate bespoke processors that support our applications running on the processor with an operating system (FreeRTOS [55]). Benchmarks are chosen to be representative of emerging ultra-low-power application domains such as wearables, internet of things, and sensor networks [73]. 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 [64]. Experiments were performed on a server housing two Intel Xeon E-2640 processors (8-cores each, 2GHz operating frequency, 64GB RAM).

Table 1: Benchmarks

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

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 each benchmark application by rewriting the RTL to remove modules that are unused by the benchmark, before performing synthesis, placement, and routing. Such an approach is representative of an Xtensa-like approach [13], 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 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.

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.

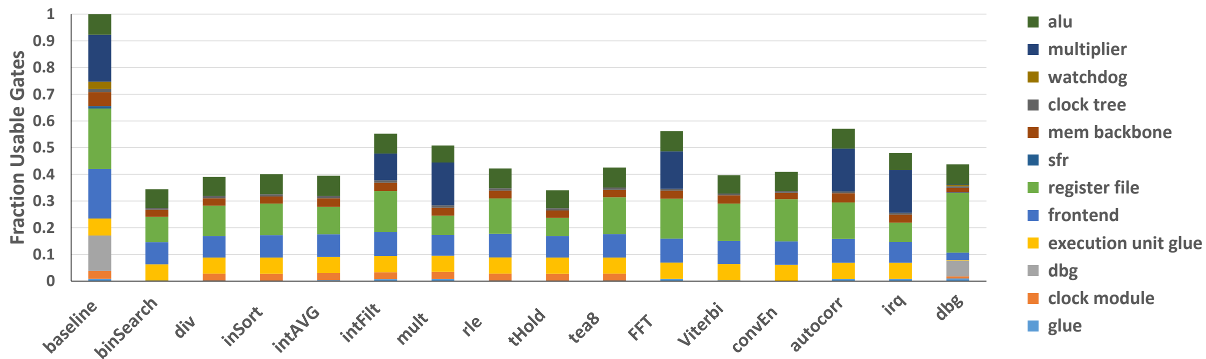


Figure 10: The height of a bar represents the fraction of gates that can be toggled by a benchmark. Components within each bar represent each module's contribution to the fraction of gates toggled by the benchmark.

Figure 10 shows the fraction of gates in the original processor design that could be toggled by each benchmark.[4] The components within each bar represent each module's contribution to the fraction of gates that can be toggled by the benchmark. The first bar in the figure shows each module's contribution to the total gates in the baseline design. We observe that each benchmark can toggle only a relatively small fraction of the gates in the baseline design. At most, 57% of the gates in the baseline design can 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 11 shows the reduction in gates, area, and power afforded by bespoke processors tailored to each benchmark. `FFT`, which has the smallest gate count reduction (44%)[5], still reduces area by 47% and power by 37%, relative to

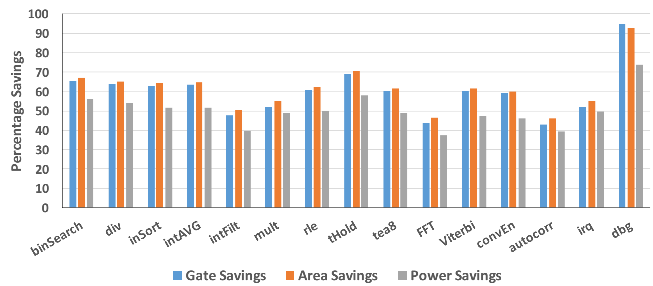


Figure 11: Reduction (%) in gate count, area, and power for a bespoke design, compared to the baseline processor.

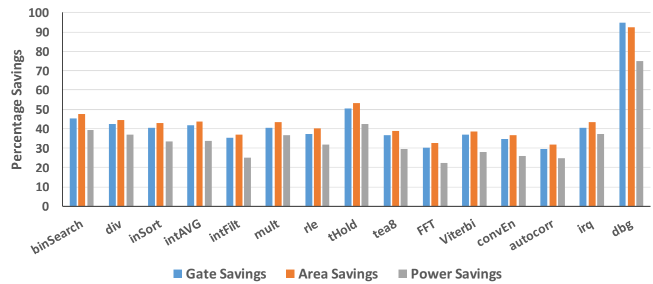


Figure 12: Reduction (%) in gate count, area, and power for bespoke designs, compared to application-specific coarse-grained module-level bespoke design.

the baseline design. Area savings are up to 92% (`dbg`), while power savings are up to 74% (`dbg`).

Figure 10 shows that some modules could be wholly removed for specific benchmarks (e.g., the multiplier can be removed for `binSearch`, since it cannot use any gates in the multiplier). For such modules, it is possible to use an Xtensa-like approach [13], 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 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.

[4]Unlike Figure 2, which presents results from profiling, Figure 10 shows results from input-independent gate analysis.

[5]Note that gate count reduction reported in Figure 11 is different than fraction of toggled gates in Figure 10, since bespoke design also removes some toggled gates that cannot propagate their toggles to state elements or output ports.

Table 2: Benefits of exploiting timing slack created by cutting, stitching, and re-synthesis.

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

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 2 shows timing slack exposed during bespoke processor tailoring for each benchmark. Exposed timing slack can be used to reduce the operating voltage of the processor without reducing the frequency.[6] Table 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.

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[7] 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 3.4.

[6]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.

[7]Table 3 shows that the runtime of X-based simulations is within an order of magnitude of a single input-based simulation.

Table 3: Verification runtime and coverage.

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

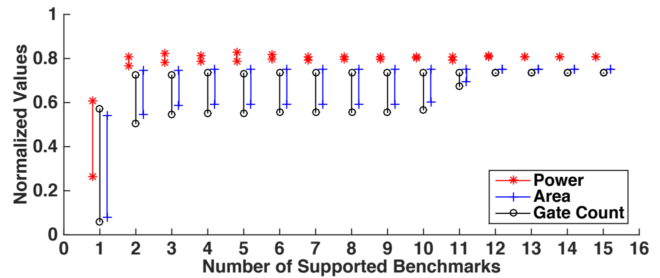


Figure 13: Normalized gate count, area, and power ranges for all possible bespoke processor supporting multiple applications.

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) [9] to generate inputs that exercise as many control paths through the application as possible. Table 3 lists the number of inputs simulated for each benchmark and the corresponding coverage of 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.[8] Table 3 also shows the aggregate runtime of the input-based simulations, providing some quantification of verification effort.[9]

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 13 shows gate count, power, and area for bespoke processors

[8]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.

[9]Note that simulations for multiple inputs can easily be parallelized to reduce the verification time significantly.

Table 4: Milu produces three types of mutants. Type I: Logical conditional operator mutants. Type II: Computation operator mutants. Type III: loop conditional operator mutants.

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

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

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.) [33]. To emulate in-field updates to fix bugs, we use the Milu mutation testing tool [38] to generate "updates" corresponding to bug fixes. Table 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., $A||B \rightarrow A\&\&B$). Type II mutants are computation operator mutants (e.g., $A+B \rightarrow A \times B$). Type III mutants are loop conditional operator mutants (e.g., $i < 32 \rightarrow i \neq 32$).

Table 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

Table 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.

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

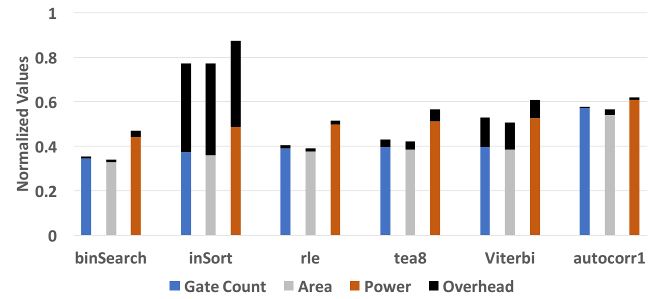


Figure 14: Normalized gate count, area, and power vs the baseline design for designs supporting all mutants (in-field updates).

bespoke processor tailored to the original "buggy" application. We see that between 25% and 100% of various mutants are covered, and 70% of all mutants are covered. This shows that a bespoke processor will maintain some of the original general purpose processor's ability to support in-field updates. If a higher coverage of possible bugs is desired, the automatically-generated mutants can be considered as independent programs while tailoring the bespoke processor for the application (see Section 3.5). Figure 14 shows the increase in gate count, area, and power required to tailor a bespoke processor to the six benchmarks with the most mutants by including all possible mutants (i.e., bug fixes) generated by Milu during bespoke design. Providing support for simple in-field updates incurs a gate count overhead of between 1% and 40%. Despite this increase in gate count, total area benefits for the bespoke processors are between 23% and 66%, while total power benefits are between 13% and 53%. Therefore, simple in-field updates can be supported while still achieving substantial area and power benefits.

A bespoke processor tailored to a specific application can be designed to support arbitrary software updates by designing it to support a Turing-complete instruction (e.g., `subneg`) or set of instructions, in addition to other programs it supports (Section 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

Table 6: Microarchitectural features in recent embedded processors

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

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

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 [17, 61][10], 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 [55]). 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 GENERALITY AND LIMITATIONS

We target bespoke processors for applications with ultra-low area and 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 [21, 46, 51, 65, 72]. 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 6). This makes our symbolic simulation-based technique a good fit for such processors. Below, we discuss how our technique may scale for complex processors and applications, if necessary.

More complex processors contain more performance-enhancing features such as large caches, prediction or speculation mechanisms, and out-of-order execution, that introduce non-determinism into the instruction stream. Co-analysis is capable of handling this added non-determinism at the expense of analysis tool runtime. For example, by injecting an X as the result of a tag check, both the cache hit and miss paths will be explored in the memory hierarchy. Similarly, since co-analysis already explores taken and not-taken paths for input-dependent branches, it can 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.

[10] Many embedded processors provide bare-metal development toolchains [56, 66].

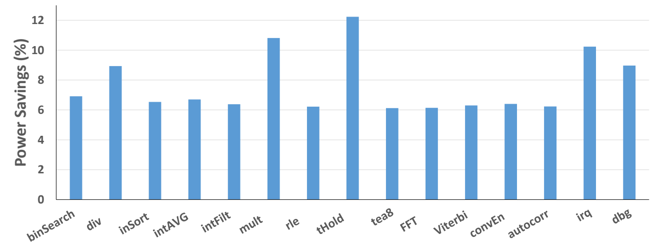


Figure 15: Power savings achieved by oracular power gating with no overheads are significantly lower than those achieved by bespoke processors for the same applications, even when each module is allowed a separate power domain and a wake-up latency of 0 is assumed.

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 paper), 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; a large number of such heuristics have been proposed [10, 32].

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 profile of unusable gates. Similarly for self-modifying code, the set of usable gates for the processor is chosen as the union of usable gate sets for all code versions. In case of fine-grained execution, 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 conservative coverage of usable gates for the thread, irrespective of the behavior of the other threads.

7 RELATED WORK

7.1 Power Gating

In this paper, we propose a method to reduce the area and power of applications running on a processor by removing unusable gates. Another 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 [2–4, 14, 19, 39, 40, 45, 52, 57, 60, 67, 71]. Some power gating techniques can even power down unused uncore modules (e.g., on-chip routers [15, 16, 20]) or dynamically re-size microarchitectural structures to better suit an application [43, 52]. 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% [58]) 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 5. Figure 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 wake-up 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 reduction 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 wake-up 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.

7.2 High-Level Synthesis

High-Level Synthesis (HLS) tools such as Cadence Stratus [12] and Mentor Catapult [47] 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.

7.3 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 [68], QsCores [69], and GreenDroid [28] 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 [29], 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 [13], 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 5) and require less manual design effort.

Chip generators [59] 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 determining 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 [35] 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-grined bespoke processor tailoring (Figure 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.

8 CONCLUSION

In this paper, 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 programmability 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%.

ACKNOWLEDGMENTS

The authors would like to thank James Myers and the anonymous reviewers for helpful suggestions and feedback.

REFERENCES

[1] 43oh. 2012. Products with an MSP430. http://43oh.com/2012/03/winner-products-using-the-msp430/. (2012).

[2] A. Abdollahi, F. Fallah, and M. Pedram. 2005. An effective power mode transition technique in MTCMOS circuits. In *Design Automation Conference, 2005. Proceedings. 42nd*. 37–42. https://doi.org/10.1109/DAC.2005.193769

[3] Abhinav Agarwal and Arvind. 2013. Leveraging Rule-based Designs for Automatic Power Domain Partitioning. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 326–333. http://dl.acm.org/citation.cfm?id=2561828.2561895

[4] Mohab Anis, Mohamed Mahmoud, Mohamed Elmasry, and Shawki Areibi. 2002. Dynamic and Leakage Power Reduction in MTCMOS Circuits Using an Automated Efficient Gate Clustering Technique. In *Proceedings of the 39th Annual Design Automation Conference (DAC '02)*. ACM, New York, NY, USA, 480–485. https://doi.org/10.1145/513918.514041

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

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

[7] Randal E. Bryant. 1990. Symbolic Simulation – Techniques and Applications. In *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*. 517–521.

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

[9] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[10] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

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

[12] Cadence. *Stratus High-Level Synthesis User Guide*. http://www.cadence.com/

[13] Cadence. 2017. Tensilica Customizable Processors. http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable. (2017).

[14] B.H. Calhoun, F.A. Honore, and A. Chandrakasan. 2003. Design methodology for fine-grained leakage control in MTCMOS. In *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*. 104–109. https://doi.org/10.1109/LPE.2003.1231844

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

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

[17] Steven Cherry. 2013. Hacking Pacemakers. http://spectrum.ieee.org/podcast/biomedical/devices/hacking-pacemakers/. (2013).

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

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

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

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

[22] Embedded Microprocessor Benchmark Consortium. 2017. EEMBC. http://www.eembc.org. (2017).

[23] Dave Evans. 2011. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. (April 2011).

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

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

[26] William F Gilreath and Phillip A Laplante. 2003. *Computer Architecture: A Minimalist Perspective*. Vol. 730. Springer Science & Business Media.

[27] O Girard. 2013. OpenMSP430 project. *available at opencores.org* (2013).

[28] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, and others. 2011. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro* 31, 2 (2011), 86–95.

[29] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (Sept. 2012), 38–51. https://doi.org/10.1109/MM.2012.51

[30] G. Hackmann, Weijun Guo, Guirong Yan, Zhuoxiong Sun, Chenyang Lu, and S. Dyke. 2014. Cyber-Physical Codesign of Distributed Structural Health Monitoring with Wireless Sensor Networks. *Parallel and Distributed Systems, IEEE Transactions on* 25, 1 (Jan 2014), 63–72. https://doi.org/10.1109/TPDS.2013.30

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

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

[33] Sönke Holthusen, Sophie Quinton, Ina Schaefer, Johannes Schlatow, and Martin Wegner. 2016. Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates. *arXiv preprint arXiv:1606.00504* (2016).

[34] IC Insights. 2017. Microcontroller Sales Regain Momentum After Slump. www.icinsights.com/news/bulletins/Microcontroller-Sales-Regain-Momentum-After-Slump. (2017).

[35] Ali Irturk, Janarbek Matai, Jason Oberg, Jeffrey Su, and Ryan Kastner. 2011. Simulate and eliminate: A top-to-bottom design methodology for automatic generation of application specific architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 8 (2011), 1173–1183.

[36] ITRS. 2015. International Technology Roadmap for Semiconductors 2.0 2015 Edition Executive Report. http://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_itrs/. (2015).

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

[38] Y. Jia and M. Harman. 2008. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*. 94–98. https://doi.org/10.1109/TAIC-PART.2008.18

[39] Y. Kanno, H. Mizuno, Y. Yasu, K. Hirose, Y. Shimazaki, T. Hoshi, Y. Miyairi, T. Ishii, Tetsuy. Yamada, T. Irita, T. Hattori, K. Yanagisawa, and N. Irie. 2007. Hierarchical Power Distribution With Power Tree in Dozens of Power Domains for 90-nm Low-Power Multi-CPU SoCs. *Solid-State Circuits, IEEE Journal of* 42, 1 (Jan 2007), 74–83. https://doi.org/10.1109/JSSC.2006.885057

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

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

[42] A. Kolbi, J. Kukula, and R. Damiano. 2001. Symbolic RTL simulation. In *Design Automation Conference, 2001. Proceedings*. 47–52. https://doi.org/10.1109/DAC.2001.156106

[43] Vasileios Kontorinis, Amirali Shayan, Dean M. Tullsen, and Rakesh Kumar. 2009. Reducing Peak Power with a Table-driven Adaptive Processor Core. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/1669112.1669137

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

[45] Changbo Long and Lei He. 2003. Distributed Sleep Transistor Network for Power Reduction. In *Proceedings of the 40th Annual Design Automation Conference (DAC '03)*. ACM, New York, NY, USA, 181–186. https://doi.org/10.1145/775832.775879

[46] Michele Magno, Luca Benini, Christian Spagnol, and E Popovici. 2013. 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*. IEEE, 189–195.

[47] Mentor Graphics. 2016. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls. (2016).

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

[49] K. Myny, E. van Veenendaal, G. H. Gelinck, J. Genoe, W. Dehaene, and P. Heremans. 2011. An 8b organic microprocessor on plastic foil. In *2011 IEEE International Solid-State Circuits Conference*. 322–324. https://doi.org/10.1109/ISSCC.2011.5746337

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

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

[52] Paula Petrica, Adam M. Izraelevitz, David H. Albonesi, and Christine A. Shoemaker. 2013. Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 13–23. https://doi.org/10.1145/2508148.2485924

[53] Gil Press. 2014. Internet of Things By The Numbers: Market Estimates And Forecasts. *Forbes* (2014).

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

[55] Real Time Engineers Ltd. 2016. The FreeRTOS website. http://www.freertos.org/. (2016).

[56] Miro Samek. 2007. Building BareMetal ARM systems with GNU. http://www.state-machine.com/arm/Building_bare-metal_ARM_with_GNU.pdf. (2007).

[57] Ashoka Sathanur, Antonio Pullini, Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. 2007. Timing-driven Row-based Power Gating. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design (ISLPED '07)*. ACM, New York, NY, USA, 104–109. https://doi.org/10.1145/1283780.1283803

[58] Naomi Seki, Lei Zhao, Jo Kei, Daisuke Ikebuchi, Yu Kojima, Yohei Hasegawa, Hideharu Amano, Toshihiro Kashima, Seidai Takeda, Toshiaki Shirai, and others. 2008. A fine-grain dynamic sleep control scheme in MIPS R3000. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. IEEE, 612–617.

[59] Ofer Shacham. 2011. *Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms*. Ph.D. Dissertation.

[60] Youngsoo Shin, Jun Seomun, Kyu-Myung Choi, and Takayasu Sakurai. 2010. Power Gating: Circuits, Design Methodologies, and Best Practice for Standard-cell VLSI Designs. *ACM Trans. Des. Autom. Electron. Syst.* 15, 4, Article 28 (Oct. 2010), 37 pages. https://doi.org/10.1145/1835420.1835421

[61] A. Silberschatz, P. Galvin, and G Gagne. 2017. Bare Machine, Wikipedia. http://en.wikipedia.org/wiki/Bare_machine. (2017).

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

[63] Synopsys. *Formality User Guide*. http://www.synopsys.com/

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

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

[66] Texas Instruments. 2015. StarterWare. http://processors.wiki.ti.com/index.php/StarterWare. (2015).

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

[68] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 205–218. https://doi.org/10.1145/1736020.1736044

[69] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. 2011. QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-specific Cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/2155620.2155640

[70] Wikipedia. 2016. List of wireless sensor nodes. (2016). https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes [Online; accessed 7-April-2016].

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

[72] Ross Yu and Thomas Watteyne. 2013. Reliable, Low Power Wireless Sensor Networks for the Internet of Things: Making Wireless Sensors as Accessible as Web Servers. *Linear Technology* (2013). http://cds.linear.com/docs/en/white-paper/wp003.pdf

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

[74] Y. Zhang, Z. Chen, and J. Wang. 2012. Speculative Symbolic Execution. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. 101–110. https://doi.org/10.1109/ISSRE.2012.8