Time Squeezing for Tiny Devices

Yuanbo Fan*[†] , Simone Campanoni*[‡], Russ Joseph[‡]

[†]NVIDIA [‡]Northwestern University

ABSTRACT

Dynamic timing slack has emerged as a compelling opportunity for eliminating inefficiency in ultra-low power embedded systems. This slack arises when all the signals have propagated through logic paths well in advance of the clock signal. When it is properly identified, the system can exploit this unused cycle time for energy savings. In this paper, we describe compiler and architecture co-design that opens new opportunities for timing slack that are otherwise impossible. Through cross-layer optimization, we introduce novel mechanisms in the hardware and in the compiler that work together to improve the benefit of circuit-level timing speculation by effectively squeezing time during execution. This approach is particularly well-suited to tiny embedded devices. Our evaluation on a gate-level model of a complete processor shows that our co-design saves (on average) 40.5% of the original energy consumption (additional 16.5% compared to the existing clock scheduling technique) across 13 workloads while retaining transparency to developers.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Embedded systems*.

KEYWORDS

code generation, timing speculation, timing slack

ACM Reference Format:
Yuanbo Fan*[†] , Simone Campanoni*[‡], Russ Joseph[‡]. 2019. Time Squeezing for Tiny Devices. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322268

1 INTRODUCTION

The next generation of tiny devices including smart city sensors, nano-drones, wearable devices, implantable electronics, and wireless sensor nodes demand reasonable performance with ultra-low energy profiles [4, 6, 14, 17, 18, 48, 50]. As we putter towards the end of Moore's Law ([44, 46]), technology scaling challenges and exhaustion of circuit-level design tricks will make these goals increasingly difficult to achieve ([2, 3]) for emerging low cost Internet of Things (IoT) applications. While this era has ushered in novel opportunities for accelerators ([33]), tiny devices will likely continue to rely on low-power general purpose processors for most of their compute needs because of their highly constraint engineering costs and low time-to-market [39]. Unfortunately, IoT systems require higher energy efficiency from low-power general purpose processors to become ubiquitous [29]. But, as opposed to costly high-performance systems which feature a wide variety of underutilized resources and heavy energy costs associated with the memory system ([32]), increasing energy efficiency in already lean low-power processors remains important and difficult. In other words, there is no easy path in front of us.

Dynamic timing slack (DTS) has emerged as a promising target for energy optimization in tiny devices. DTS refers to the portion of the clock period that remains after all the signals have propagated through logic paths (Figure 1 highlights this concept). When data waits on the latch/flip-flop inputs well in advance of the clock edge, this interval can be viewed as wasted time. State-of-the-art low-power architectures [16] are able to detect the presence of this slack within a single clock cycle and they scale down the supply voltage to reclaim energy. These approaches are already able to save significant energy (between 10% and 20% of the processor power, besides PVT margin reduction), but they are all limited by having a compiler that generates code without being aware of DTS. We argue that a DTS-aware code generation is currently the missing piece. This paper shows that relatively simple changes to an industrial-strength compiler tailored to a DTS-optimized low-power processor almost doubles the energy savings of such processor (from 24% energy savings gained by a DTS-unaware code generation to 40.5%).

DTS (and therefore the energy savings enabled by it) in IoT processors are mostly limited by the critical paths in the execution units

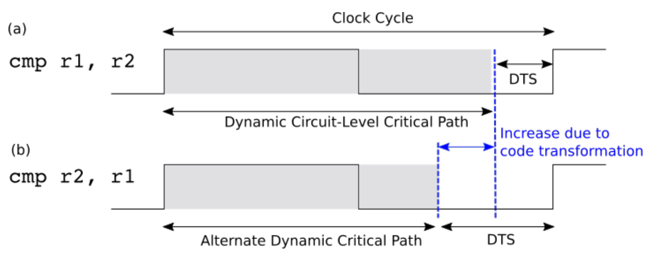

(a)
Clock Cycle
cmp r1, r2
DTS
Dynamic Circuit-Level Critical Path
(b)
cmp r2, r1
Increase due to code transformation
Alternate Dynamic Critical Path
DTS

Figure 1: Effects of code transformation and clock scheduling on clock cycle.

*Yuanbo Fan and Simone Campanoni contributed equally to the research presented in this paper.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6669-4/19/06...$15.00
https://doi.org/10.1145/3307650.3322268

– the most prominent being the carry chain within ALUs and effective address generators. From the carry chain, these paths typically extend to condition generation logic and register bypass networks. Prior work has identified timing slack within the execution units of server chips [35] as well as ultra-low power microcontrollers [7]] and we have witnessed similar behavior on our own testbed. Our observation that motivated us to design, build, and test the work described in this paper is that most carry chain stress is triggered by code patterns that a compiler has full control over. In more detail, most carry chains are triggered by subtracting small values from another one. Often, these subtractions relate to operations that a compiler has full control over, like effective address computation. For example, computing the effective address of a memory location by subtracting a small offset value (e.g., 4) from a large value like a stack base address (e.g., `0xFFFFFF40`) forces the hardware both to invert the former and to add one due to two's complement representation. Then, the hardware performs the add operation using the so-generated negative value (e.g., `0xFFFFFF40 + (-4)`). The hardware is forced to perform the inversion because of the two's complement representation used by all commodity processors. The inversion of a small value propagates from the carry-in bit the entire way through the fast adder structures and beyond. This limits DTS and, therefore, blocks the energy savings that a DTS-optimized low-power processor could otherwise obtain. Notice that the same task can be accomplished by adding a small offset (e.g., 4) from a different base address (e.g., `FFFFFF3C`), which leads to much less (often none) carry chain stress that significantly increases its DTS. Finally, while this seems a simple change, it requires significant modifications to the code generator of a compiler to compute the effective memory addresses following this new scheme. This also necessitates non-trivial changes to the memory data layout as well. Note that this is only one example of the DTS-aware code generation that our compiler performs.

In this paper, we introduce co-design which permits cross-layer optimization of the compiler and architecture to exploit DTS in novel ways on tiny devices. The overarching innovation is the recognition that program data fuels many of the opportunities for timing slack and that by elevating timing models into the compiler, we can create additional DTS by squeezing operations at a sub-cycle resolution and enhancing the architecture to benefit from these opportunities. Previous work has viewed slack without regard to the way that the presence or absence of certain types of data values or placement of data impact timing slack. We demonstrate that a compiler can leverage its ability to manipulate both code generation and data layout to create new types of timing slack. This approach is well-suited to tiny devices where there are limited opportunities for additional energy savings and the designer has significant control of the system stack allowing for aggressive cross-layer optimizations. We demonstrate our approach by fully implementing it in a modern industrial-strength compiler and evaluating it on a gate-level model of a ultra low-power processor, which has been validated against a fabricated chip.

This paper makes the following contributions: (i) We show that current DTS of state-of-the-art low-power processors is limited by code patterns that compilers have full control of (ii) We designed the first DTS-aware compiler which considers the impact that data has on timing slack. The compiler uses this knowledge to temporally squeeze operations to expose an additional amount of dynamic timing slack to the hardware. (iii) We demonstrate the great potential of coupling DTS-aware compilers and architecture to save significant energy on tiny devices by targeting dynamic timing slack on at a fine temporal granularity. (iv) We depict simple but powerful hardware support for effective address calculation which allows the system to bypass an otherwise problematic critical path.

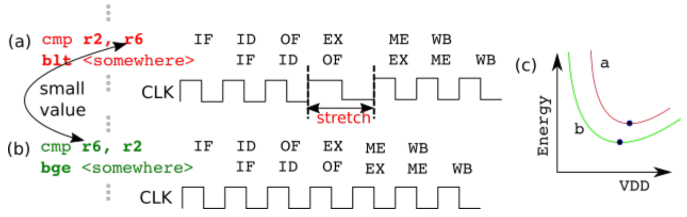


Figure 2: (a) Critical path stressing instruction requires clock stretch. (b) Compiler can generate code which avoids critical path. (c) System-wide energy savings enabled by VDD scaling at the same performance.

2 BACKGROUND AND MOTIVATION

Dynamic timing slack (DTS) refers to the underutilized fraction of the clock cycle during which data has already propagated through all logic paths and waits to be latched on the subsequent clock edge. Figure 1a visually shows how timing slack emerges for a `cmp` instruction. When the result produced by the `cmp` instruction is available well before the clock edge, there is dynamic timing slack (presuming that none of the other critical paths are exercised).

Like many other types of slack found in the system, these idle intervals can be viewed as an opportunity. Persistent and identifiable timing slack could be traded for either performance improvement by increasing the clock frequency or energy savings by lowering the supply voltage. Timing speculative architectures like Razor, EVAL, or CLK-Sched ([15, 16, 42]) are built to function at marginal operating conditions and can directly exploit existing DTS to improve energy-efficiency without any additional design effort. These systems rely on error detection and correction logic to identify errors and guarantee architecturally correct results.

This work is motivated by our observation that there are deep connections between the DTS and the workload, which are not understood or controlled by today's compilers. For example, the DTS of the cmp instruction of Figure 1a is sensitive to the operand order. A different order can lead to higher DTS (and therefore higher energy savings) as shown in Figure 1b. This is because the hardware inverts the second operand of a `cmp` to perform the comparison. The latency of such inversion depends on the value stored in the second operand. Today's compilers ignore this (and other) aspects of the code being generated and, therefore, they often generate code with high circuit-level latencies, which limits the energy saved by the underlying DTS-optimized architecture. On the other hand, a DTS-aware compiler can decide the order of the operands based on both the understanding of the possible value ranges of the operands of an instruction and their related DTS. This is an example that motivates the need for the code generation of a compiler to become DTS-aware. This is the topic of this paper.

2.1 Timing Slack in Arithmetic Units

All of the timing in CMOS logic paths, including those responsible for producing DTS, are heavily influenced by data placed on the inputs. In the case of a processor pipeline, the quantity of timing slack is a function of the circuit structure and the instructions (and their data) currently executing. Via logical structures in the circuit, input data patterns will activate gate switching on various portions of the fanout tree, causing different timing paths to appear. The exact data sequencing that causes the DTS to be present depends on the circuit-level implementation of the logic.

For DTS on tiny devices, the adder carry chain will likely play an out-sized role due to its status as a critical path structure. Previous empirical studies on commercially available server-class processors has shown timing slack in compute units [35]. Addition is a basic atomic operation which is required to finish within one cycle to maintain good performance for many operations. There are many design consideration in construction of a fast carry look ahead adder [36], and length of the carry chain figures heavily in the performance of the unit. Some recent adder designs have even relied on speculative carry-in values to hide carry chain latency [20, 31, 47].

Beyond its prominence as a circuit-level timing critical operation, adder hardware is heavily utilized because the addition primitive is prominent in most instruction sets. Addition is used in various arithmetic computations, branch target address calculation, effective address generation for memory operations, and value comparison. The combination of timing criticality and heavy utilization make addition a natural target for optimization. While commercial designs typically have large numbers of critical paths, it is important to note that path activity also plays a large role in the availability of dynamic timing slack. If many of these other critical paths are connected to edge cases or infrequent events, they would have low activity and hence would not set a lower bound on the available slack.

Each of these prominent adder use cases place different collections of input patterns on the adder hardware, leading to different degrees of stress on critical paths and distinct types of DTS. Without loss of generality, adder inputs can be viewed as two unsigned integers and a carry in signal. Via two's complement representation, signed values and subtraction/comparison are also supported. Due to gate-level switching, various sequences of inputs would lead to different amounts of activity on the carry chain. For instance, a stream of small magnitude values would neither generate or propagate signals that would stress the long carry chain. This means that DTS may be present when small magnitude values are added. On the other hand, a mixture of large and small magnitude values would have a much greater chance of stressing the carry chain. This is particularly relevant in the case of subtraction and comparison with small magnitude values. With two's complement representation, a likely outcome is that large and small magnitude values appear on the adder and consequently stress the carry chain. As a result, these types of operations do not create significant timing slack.

2.2 Prior Work

Recent work has examined ways to exploit dynamic timing slack through a number of means that use a combination of hardware mechanisms and varying amounts of compiler assistance as summarized in Table 1. Compiler support for timing speculation was examined in early work by Hoang [24] who studied opcode substitutions to reduce activity on critical paths and Satori [43] who evaluated several standard compiler optimizations to understand their impact on timing speculation. More recently, researchers have identified ways to exploit timing slack in a non-speculative manner [7, 9]. The introduction of fine-grained clock scaling allows the system to recover more DTS [9, 16] by adapting the frequency to meet the computational needs of individual instructions. The CLK-SCHED approach proposed in [16] takes this to an extreme by allowing the compiler to embed clock control directives within binaries on systems that support timing speculation. These directives control a fast PLL which adapts the clock phase to take advantage of timing slack at a sub-cycle resolution. The presence of speculation allows the compiler to aggressively scale the clock frequencies to achieve larger DTS benefits. All of these approaches however ignore the role that data values themselves play on DTS. In addition, they do not address the ways in which data placement impacts the DTS in the system. The TimeSqueezer Compiler presented in this paper identifies this as an opportunity and applies data aware compilation and data placement for timing speculative IoT processors.

Approach	Hoang [24]	Satori [43]	Cherupalli [7]	DynOR [9]	CLK-Sched [16]	TimeSqueezer
Speculative	✓	✓	✗	✗	✓	✓
Fine Grained	✗	✗	✗	✓	✓	✓
Clock Adjustment						
Code Transformation	✓	✓	✗	✗	✗	✓
Data Aware	✗	✗	✗	✗	✗	✓
Data Placement	✗	✗	✗	✗	✗	✓

Table 1: Comparing ways of exploiting dynamic timing slack.

2.3 Compiler Influence

Attention to the types of data flowing through the pipeline helps to identify situations where timing slack may or may not be present. From a programmer's point of view, subtraction and comparison operations seem fairly benign, and it seems unreasonable to expect a developer to re-factor code to avoid these types of instructions. At the same time, much of the function of the compiler is structuring code and re-organizing the program to maximize the capabilities of the underlying hardware. Given knowledge of what are critical path stressing operations, the compiler is well-equipped to generate code that creates more dynamic timing slack by substituting code that places less stress on critical paths for example. Together Figure 1a and 1b show an example of how differences in code can impact timing slack. Figure 2 shows how code transformations can drastically improve the energy profile of a system – additional DTS allows the system to aggressively scale the voltage to achieve greater energy savings. More broadly speaking, co-designing the compiler with the architecture enables new optimizations that expand possibilities for generating and managing timing slack. The key component is

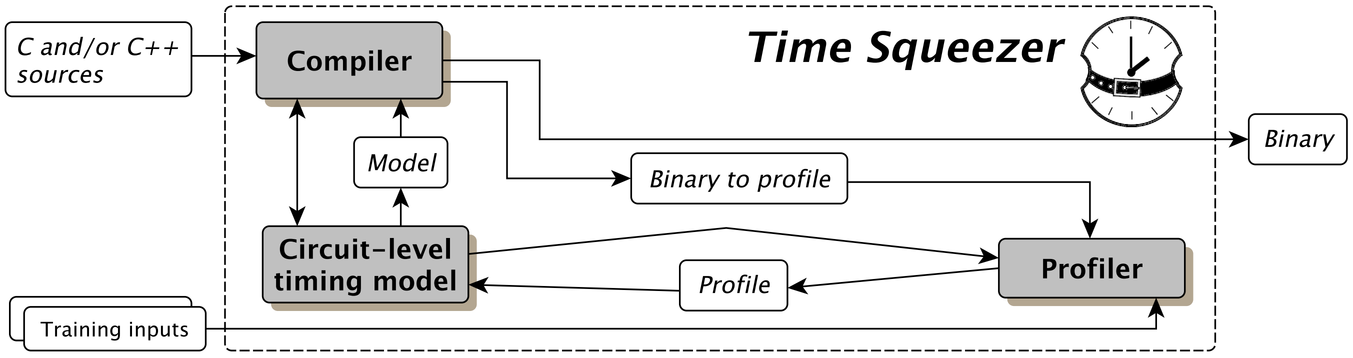


Figure 3: Overview of the Time Squeezer system.

to understand the impact that data has on the timing paths. This has been a relatively overlooked in previous work where the focus has been on opcodes associated with timing critical operations. In this paper, we focus on the use of adder logic for two extremely common operations: effective address calculation and integer value comparison.

3 THE TIME SQUEEZER SOLUTION

The Time Squeezer system is the first compiler-architecture co-design able to coordinate the generation of code and placement of data to reduce the DTS of the processor pipeline to save energy. To this end, the compiler performs code generation and memory alignment minimizing the resulting DTS. The underlying architecture assumes such DTS-aware code to aggressively adjust the clock cycle to the dynamic needs of the workload. This leads to an energy efficient system without compromising performance or output quality as well as without the need for developer assistance.

Adjusting the clock cycle The Time Squeezer compiler generates instructions to drive the clock cycle of the underlying architecture as in [16]. The timing speculative hardware automatically detects and recovers from potential run-time errors guaranteeing correctness independently to the clock cycles requested by the code. This clock management scheme has been proposed by previous work and can be considered mature [10, 26]. The work described in this paper uses this scheme as an enabler for our co-design.

Code and DTS: a tight relation More timing slack means more opportunities to lower supply voltage. Due to the strong connection between instruction sequences and circuit-level timing slack, the code that is more likely to produce more timing slack is regarded as a "good" pattern. To specific hardwares, there may be various "good" or "bad" patterns. Our compiler focuses on two of the most common patterns: memory address computation and comparing values.

With a compiler and architecture co-design, the compiler generates code being aware of its circuit-level criticality, while the architecture assumes that the "good" patterns are common cases. This allows a more efficient overall design, compared to any software-only or hardware-only techniques.

4 DTS-AWARE COMPILER

Adders are used for many purposes in today's systems and are the workhorses of ALU and effective address generation units. Their most straightforward use is adding values stored in general-purpose registers. However, they are also used as building blocks for two other purposes: computing addresses needed by memory instructions (both stack and heap) and value comparison. These last two use cases are fundamentally different than the first one: one operand is typically much larger than the second one. This leads to have different DTSs for different operand orders and/or operand signs. Current compilers ignore this relation and, therefore, they generate code with sub-optimal DTS. We need a DTS-aware code generation design, which we have embedded within our compiler. The DTS-awareness of our compiler enables the underlying architecture to shrink the dynamic timing slack. To the best of our knowledge, we are the first ones to have recognized these opportunities and the first ones to have empirically evaluated their benefits. The rest of this section describes the DTS-aware components of our compiler.

4.1 DTS-Aware Stack Accesses

Stack accesses are some of the most common operations in programs, and current compilers generate the code for them without considering their DTS. The effective address computation associated with stack accesses is necessary and critical. Due to a limited number of registers, a compiler has to map most program variables to stack locations (this is called spilling and it is performed during register allocation inside a compiler back-end). This leads to frequent load and store operations from and to the stack at run time. The frequent use of stack memory requires their accesses to have a fast response time making it sensitive to timing constraints. After describing the code that current compilers generate to access the stack and its memory layout, we describe our DTS-aware design.

Conventional compilers The stack memory grows downwards from high memory addresses to low ones on commodity processors (e.g., Intel, ARM, IBM chips), as shown in Figure 4. At compile time, a compiler layouts the allocation frame of a function `f` reserving a stack location for each variable of `f` that could not be mapped to a processor register by the compiler register allocator (e.g., graph coloring). For example, Figure 4 shows the allocation frame of the hot function `susan_principle` of the `susan` benchmark. This function includes two variables, `x_size` and `y_size`, that are spilled to the stack. The code generated for `f` includes load and store instructions for accessing these spilled variables. The effective address of these

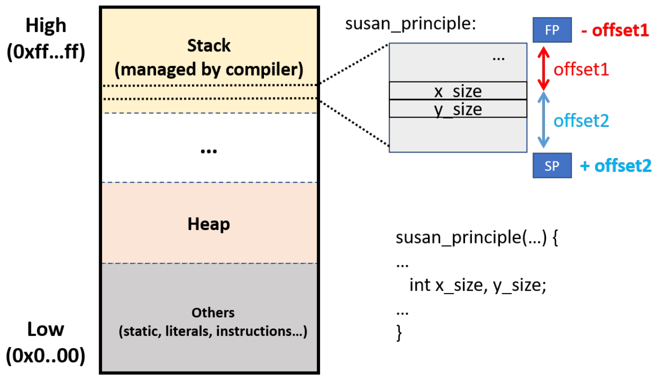
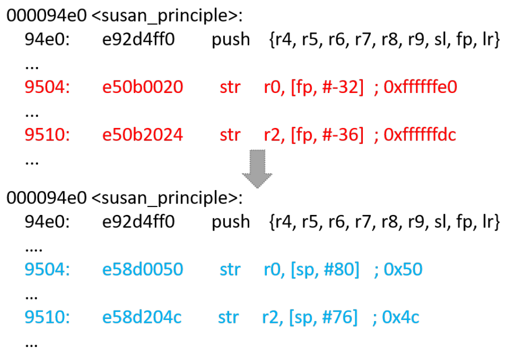


Figure 4: Stack layout of a function in susan



```
000094e0 <susan_principle>:
  94e0:    e92d4ff0    push  {r4, r5, r6, r7, r8, r9, sl, fp, lr}
  ...
  9504:    e50b0020    str   r0, [fp, #-32]  ; 0xffffffe0
  ...
  9510:    e50b2024    str   r2, [fp, #-36]  ; 0xffffffdc
  ...
000094e0 <susan_principle>:
  94e0:    e92d4ff0    push  {r4, r5, r6, r7, r8, r9, sl, fp, lr}
  ....
  9504:    e58d0050    str   r0, [sp, #80]   ; 0x50
  ...
  9510:    e58d204c    str   r2, [sp, #76]   ; 0x4c
  ...
```

Figure 5: Code transformation for a hot function in susan.

memory instructions is computed by adding a base register and an offset.

The code generated by DTS-unaware compilers (e.g., `clang`, `gcc`, `icc`) compute effective addresses of stack locations by subtracting a small offset from a large unsigned number. Conventional compilers use the address of the base of the allocation frame, called frame pointer (`fp`), of a function `f` as base register. Because the stack grows downwards, this base register includes a high unsigned number and the offset used to compute the effective address is negative. This operation is commonly implemented using a fast adder architecture and two's complement coding.

Opportunity The effective address of stack accesses are computed by subtracting a small offset from a large unsigned number using two's complement coding. The subtraction operation in an adder, therefore, requires to invert each bit of an offset and set the carry in bit to 1. This computation in adders exercises timing-critical paths, which leads to long delays. Such long-delay computation is not a problem for conventional architectures because such latencies are still lower than a single clock cycle. Therefore, this long-delay computation does not impact the overall performance of conventional architectures.

Our proposed architecture dynamically adjusts the clock cycle from its nominal period to remove the inefficiencies generated by the dynamic timing slack (DTS). This ability enables a system to reduce the DTS, but it also makes the long-delay pattern of effective address computations of stack accesses an important roadblock for additional energy efficiency: long-delay computations impact the performance for our architecture even if they are less than the nominal clock cycle. The roadblock is created by inverting an address offset required by the related subtraction operation. Removing this operation eliminates this roadblock and enables the underlying architecture to save more energy by further reducing the DTS.

Removing the need for inverting an offset can be obtained by observing that the effective address of a stack location can be accessed by *adding* a small offset to the stack pointer (`sp`), rather than *subtracting* an offset from the `fp`. For example, the spilled variable `x_size` of the function `susan_principle` of Figure 4 can be accessed by adding `80` to `sp` rather than subtracting `32` from the fp. Accessing stack locations by using the `sp` avoids the inversion of the offset and, therefore, the related long carry chain. This reduces significantly the latency of accessing stack locations, as shown in Figure 6a vs. 6b. This is what the code generated by our compiler performs. While this opportunity enables our system to gain significant energy benefits, it comes with its challenges related to the dynamic stack allocation that shifts the `sp`.

The Time Squeezer compiler The code generated by our compiler computes the effective addresses of stack locations by adding a small offset to the `sp`. Figure 5 compares the code generated by the DTS-unaware `clang` compiler and the one our DTS-aware compiler generates for the function `susan_principle` of Figure 4. This figure shows that instead of computing the effective address by removing `-32` from the register `fp`, our compiler generates the code that adds `80` to `sp`.

Changing the computation of effective addresses of stack variables reduces their latencies. This is shown in Figure 6. This figure shows that by adding a small number to a large one rather than subtracting it blocks the otherwise long carry chain. This reduces significantly the latency of this operation creating more room for the underlying architecture to shrink the DTS.

Most program functions allow our compiler to do a simple transformation like the one described above. However, some functions dynamically expand their allocation frames by using the `alloca` C library function. For these functions, the `sp` changes and, therefore, the stack variable offsets computed from it change as well. Hence, variable offsets are not known at compile time anymore and they must be stored and updated accordingly. To handle these functions, our compiler creates a new section in their allocation frames to store the offsets of their spilled variables. Moreover, our compiler generates extra code (a load, an add, and a store) to update these offsets according to the dynamic expansion. This offset-update code is added just after each invocation of `alloca`. Finally, when necessary load instructions are inserted to access the offsets of spilled variables.

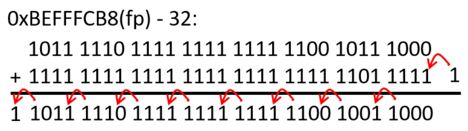
To reduce the overhead of the offset-update code, our compiler performs two optimizations. The first one targets the overhead of the offset updates. The second one targets the overhead of loading the offsets.

Optimization 1: To reduce the offset update overhead, our compiler performs the offset loads and stores in batches to reduce the overhead of multiple `alloca` invocations. This optimization removes the need for a load and a store at every `alloca` invocation. Moreover, this optimization is only enabled when the compiler can guarantee that no access of the related variable offsets is performed between the batched `alloca` invocations. This is performed by an ad-hoc data-flow analysis that tracks the accesses of the offsets of spilled variables through the control-flow graph (CFG). This ad-hoc analysis is a customization of the intra-procedural reaching definition analysis. The customization is on the definition of GEN and KILL sets and on the data flow values. Our data flow values are always either empty or they include a unique symbol S, which represents all stack variables. This is because we do not need to differentiate between stack variables: all of them need their offsets to be updated or none of them do. The GEN set of an instruction i is either S or nothing. It is S if the instruction i accesses a stack variable; it is nothing otherwise. The KILL set of an instruction is either S or nothing; it is S if the instruction is `alloca`, nothing otherwise. The IN and OUT sets are computed following the same equations of the reaching definition analysis.

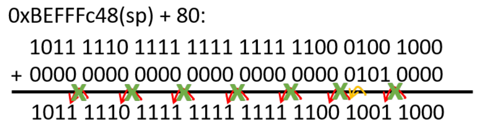
The optimization related to this data flow analysis is related to pairs of `alloca` instructions. Let us assume we have two `alloca` instructions A_1 and A_2. The goal is to avoid (when possible) updating the stack variable offsets after A_1. Our optimization considers only pairs A_1, A_2 that are control equivalent (i.e., A_1 dominats A_2 and A_2 post-dominates A_1). If IN[A_2] is empty (in other words, it does not have the symbol S), then it is guaranteed that: all paths between A_1 and A_2 have no instructions that access a stack variable: this is guaranteed by the data flow analysis; A_1 will always be executed before A_2: this is because A_1 dominates A_2; A_2 will always be executed after A_1: this is because A_2 post-dominates A_1. Hence, we can avoid updating the offsets of stack variables after A_1 and fold these A_1-updates within the ones that have to be done after A_2. The conservativeness of our data flow analysis is of the same nature of the one for reaching definition analysis: infeasible paths. In practice, however, we did not see any reduction in optimality due to this conservativeness. This is because the typical code patterns followed by developers that use `alloca` is that they invoke `alloca` within the first code block of the function; hence, no infeasible paths exist yet at that point in the code.

Optimization 2: To reduce the overhead of loading the offset of a spilled variable, our compiler avoids injecting loads when it can guarantee that the related variable offset did not change since the last offset access. This is performed by using another ad-hoc dataflow analysis. This ad-hoc dataflow analysis is similar to the previous one with the only difference in the definition of the GEN and KILL sets of instructions. The GEN set of an instruction i is either S or nothing. It is S if the instruction i is `alloca`; it is nothing otherwise. The KILL set of an instruction is either S or nothing; it is S if the instruction accesses a stack variable offset, nothing otherwise.

The optimization related to this data flow analysis is related to pairs of accesses to the same stack variable offset. Let us assume we have two instructions of this kind, I_1 and I_2. The goal of this optimization is to avoid (when possible) re-loading the stack variable offset accessed by I_2 after I_1. For this optimization, we consider only pairs I_1, I_2 such that the former dominates the latter. If IN[I_2] is empty, then we can avoid re-loading the stack offset in I_2 and use instead the one loaded by I_1.



(a) Long delay pattern when using subtraction.



(b) Short delay pattern when using addition.

Figure 6: Delay patterns

4.2 DTS-Aware Memory Alignment

To improve the efficiency of effective address calculation, our compiler aligns both the stack allocation frames and the memory objects. Our underlying architecture assumes such alignments and it computes effective addresses using bit-wise OR operations rather than the much more error-prone and energy-hungry additions.

Our compiler increases the size of the allocation frame of a compiled function `f` to make the `sp` a power of two. This alignment guarantees that the offsets of the spilled variables of `f` are encoded only using the lowest significant bits unused (they are always zero) by the `sp`. Hence, the computation of the effective address of spilled variables can be performed by using an bit-wise OR operation rather than an otherwise-needed additions. Notice that the depth of the circuit of the bit-wise OR operation is constant (1 gate delay) with the number of bits which significantly reducing the latency of the address computation, as well as saving unnecessary transitions/energy.

Similarly, our compiler aligns the objects (e.g., structures) allocated in the memory heap. To do so, our compiler substitutes calls to heap allocators to both allocate memory (e.g., `malloc`) and to free it (e.g., `free`). The former substitution is done to increase the amount of the memory requested to perform the required alignment. The latter substitution is performed to make the alignment transparent to heap allocators (e.g., the C standard library). In more detail, we allocate enough extra memory to force the alignment of the base address of an allocated object to be a power of two.

Rather than using the address returned by the heap allocator as base address of the object, the code generated by our compiler uses the lowest address within the memory allocated after CPU-word bytes that is a power of two. Similarly for the computation of stack location addresses, our underlying architecture assumes this alignment also for heap memory accesses allowing the computation of effective addresses of fields of heap objects (e.g., a field of a structure) to be performed using bit-wise OR operations rather than critical path stressing additions. Finally, to make this transformation transparent to heap allocators (e.g., the standard C library), our compiler generates additional code to properly round the address given to the free/destroy API.

4.3 DTS-Aware Value Comparisons

Another common use of adders is comparing register values. The compare instruction in commodity processors (e.g., Intel, ARM) indicates whether the first register value is less, equal, or greater than the second one. This is performed via an implied subtraction operation.

Conventional compilers The two operands of a compare instruction can be either registers or constants. If both operands are registers, today's compilers generate compare instructions keeping the order of the operands decided by their front-end, which follows the natural order defined by the source code that does not take into account architecture-specific characteristics. If an operand of a compare instruction is a constant, then today's compilers change the order of operands only if the underlying architecture requires it. For example, the `cmpq` instruction of Intel processors requires a register for the second operand (e.g., "`cmpq %rax, $10`" is an illegal `x86_64` instruction). However, ARM processors do not have this requirement.

Opportunity To compare register values, today's architectures subtract the second register value from the first one. Because two's complement coding is used, the adder inverts each bit of the second register value and set the carry-in bit to `1`. Then, the first register value is added to the second one so inverted. The latency of the subtraction operation directly depends on the propagation of the carry-in bit of the inversion performed. In more detail, the less is the propagation of the carry-in bit, the shorter is the critical-path in the circuit exercised at run-time, the lower is the sub-cycle latency of the subtraction operation.

Lower integer values are more likely to have longer propagation when being placed in the second operand. (Try $5 - 3 = 2$ and $3 - 5 = -2$, similar to Figure 6) Therefore, it is preferred to subtract higher values from lower ones. Moreover, we can swap the two registers used in a compare instruction without affecting the original semantics of the compiled program. Hence, a compiler can use this degree of freedom to place the register that is more likely to hold at run-time low integer values in the first operand of a compare instruction. This will decrease the critical-path of the inversion of the related compare instruction if the inference made by such compiler is more often valid than not. Finally, if the underlying architecture allows it, a compiler can also exercises this degree of freedom even when an operand is a constant.

The Time Squeezer compiler Our compiler generates compare instructions with a low probability of stressing the circuit-level critical paths. We achieve this goal by relying on training inputs provided by developers to construct a bit-level probabilistic representation of the values being compared. The compiler then applies architecture specific analysis to select a compare instruction where the second operand is less likely to propagate the carry-in bit when inverted. This reduces the length of the dynamic critical path and squeezes the time needed by the computation to complete and thereby increases the DTS, which is exploited by our underlying architecture to save energy.

In more detail, our compiler includes Algorithm 1 to decide whether or not it should swap the operands given by the front-end.

Algorithm 1 Input-aware delay model

```
Instruction: cmp X, Y
//Operands are seen as sequence of bits
X: $\{x_n, ...x_1, x_0\}$
Y: $\{y_n, ...y_1, y_0\}$

$Lat_i$ := Latency of propagating the carry-in bit from the bit i to
    the most significant bit
P(x, j) := Probability that the $j_{th}$ bit of x is 1

//Probability of $x_j = 1$ and $y_j = 0$
$P_{x_j,y_j=1,0} := Px, j \times 1 - Py, j$

//Probability of $x_j$ and $y_j$ having the same value
$P_{x_j=y_j} := Px, j \times Py, j + 1 - Px, j \times 1 - Py, j$

procedure BOOL should_swap_cmp_operands(x, y)
    swap = 0
    for i = 0 : N-1 do // each bit in x/y
        $P_{swap}i = P_{x_j,y_j=1,0} \times {}_{k\in\{0,i-1\}} P_{same}x, y, k$
        $P_{keep}i = P_{x_j,y_j=0,1} \times {}_{k\in\{0,i-1\}} P_{same}x, y, k$
        $L_{swap}+ = P_{swap}i \times Lat_{i+1} + 1 - P_{swap}i \times Lat_i$
        $L_{keep}+ = P_{keep}i \times Lat_{i+1} + 1 - P_{keep}i \times Lat_i$
    end for
    return ($L_{swap} < L_{keep}$)
end procedure
```

Profiling The Time Squeezer system includes a code profiler to estimate the probability that each bit of an operand used in an integer comparison is equal to `1`. The compiler then applies this information to estimate the latency of inverting each operand using architecture-specific information (i.e., the circuit structure of the adders in the targeted processor). Specifically, for a compare instruction `C` that uses a register X as an operand, our profiler computes the frequency that each bit j of X was `1` across all invocations of C for all training inputs. We denote this as Px, j in Algorithm 1. The compiler then considers two scenarios: one in which the operands keep their initial ordering and one in which the order is swapped. The compiler then estimated the most likely latency of the compare instruction for these two cases (L_{swap} and L_{keep} of Algorithm 1) using a representation of the gate-level structure of the adders that we describe next. The model essentially computes gate delays to mimic the dynamic critical path under these operands. The result of this analysis will help the compiler to determine whether or not to swap the operands of C.

Latency Our solution to compute latency is based on a conservative analysis that is both fast (it adds only a small compilation time) and data driven. We apply the profile analysis and adder-specific information to arrive at the result. The latency calculation is based on the following observation: each column-wise pair of operand bits can potentially allow or suppress the carry on its way to the most significant bit.

For example, let us assume a compare instruction `cmp x y`. If bit i of x and y are respectively 0 and 1, then the carry-in bit propagation that could have started from the lower significant bits of i is guaranteed to stop at bit i. This is because y is inverted leading its i-th bit to 0 and, therefore, the sum of two zeros and a potential carry-in bit coming from the bit $i - 1$ cannot generate a carry-in bit to propagate to $i+1$. In other words, the carry-in bit propagation is blocked at bit i.

Now, let us consider the opposite scenario: the bit i of x and y are respectively 1 and 0. In this case, we do not have the same guarantee and the carry-in bit propagation could go through the bit i leading to higher sub-clock latencies. However, we can swap the two operands generating the instruction `cmp y x` bringing back the more profitable scenario described above. In the later case, swapping the two operands has reduced the instruction latency. The exact amount depends on the circuit-level construction of the the adders in the underlying architecture (this is encoded in Lat_i of Algorithm 1).

At a high-level, the latency computation applies profile data to determine under which circumstances the ordering of operands will be likely or unlikely to stop carry propagation and thereby whether or not it stresses the critical path. Based on the above discussion, we can consider $P_{swap}i$ as the probability that we benefit from swapping the original operands based on their i-th bit and that the i-th bit is the lowest significant bit that x and y differ. In other words, we cannot decide whether or not we should swap x and y based on the lower bits of i.

We next compute the latency of the compare instruction given as input when we swap its operands by combining the likelihood that the need for an operand swap occurs (i.e., $P_{swap}i$) with the latency of the instruction created by the swap. In this computation, Lat_{i+1} represents the longest circuit-level critical path obtained by starting the carry-in bit propagation at bit $i+1$. We add to this latency the one obtained if the need for an operand swap does not occur. In this case, the worst latency starting at bit i is Lat_i.

Similarly, the latency of the instruction when we keep the original order of operands is computed. Finally, we accumulate all latencies across all bits as each one could block the carry-in propagation. The compiler decides whether or not to swap the operands based on the final accumulated latencies (L_{swap} and L_{keep}) in line 28 of Algorithm 1.

4.4 The Case for a Hw/Sw Co-Design

The transformations described in this section are motivated by co-designing our compiler with the underlying architecture. In other words, they should not be used for conventional architectures.

The first transformation, computing effective addresses of stack locations using `sp` rather than `fp`, can lead to unjustified overhead for conventional architectures. This overhead comes from updating at run-time stack variable offsets of functions that expand dynamically their allocation frames. This overhead, however, is not justified for conventional architectures that cannot take advantage of the extra DTS generated by this transformation. In more detail, conventional compilers rely on `fp` to compute stack variable offsets, which is constant during the execution of an invocation of the related function. Therefore, no run-time updates are needed for stack variable offsets (this is the reason why conventional compilers rely on `fp` rather than `sp`). While our transformation generates a small overhead for these functions, it creates significantly more room for shrinking the DTS. Our architecture takes advantage of this extra DTS generating an overall important energy saving. On the other hand, conventional architectures do not. Therefore, our transformation should only be used for the co-designed architecture described in the next section.

Similarly, the stack and heap alignment transformations described in this section create additional memory consumptions, which are unjustified for conventional architectures. These transformations increase (even if slightly) the memory consumption of the compiled programs to align the base address of the allocated heap objects. However, conventional architectures are not able to take advantage of their benefits of shrinking the sub-cycle latencies of the heap address computations.

Finally, the last transformation described in this section (value comparison swap) has no run-time effect to conventional architectures. Its use would only increase the compilation time for such architectures.

5 DTS-OPTIMIZED ARCHITECTURE

Prior work proposed fine-grained clock management schemes to adjust clock period at cycle-level [9, 26]. Time Squeezer builds on top of the previous work [9, 26]. The hardware makes heavy use of timing speculation and ultra fine-grained clock frequency scaling.

Timing Speculation The architecture operates in an overscaled supply voltage mode and relies on error checking and recovery to ensure architecturally correct execution. In our proposed design, timing violations are detected and corrected using the well-known RazorII mechanism [12]. Note that our approach is completely agnostic to the checker, and it would be possible to substitute any other mechanism that supports timing speculation.

Clock Control We integrated a multi-phase All-Digital Phase-Locked Loop (ADPLL) tightly with the pipeline design. In the ADPLL design, there are multiple equally-spaced phases generated from the Digital Controlled Oscillator (DCO), which enables real-time clock period modulation. As different phases are selected, the frequency of the PLL remains the same, which allows cycle-by-cycle clock period adjustment. The ADPLL stretches the clock period for instructions that are likely to trigger errors before they trigger their critical path, therefore avoiding runtime errors and the corresponding the recovery cost. The multi-phase ADPLL is programmed by our compiler, which inserts directives into the static instruction stream based on instruction-level delay models, as introduced in [16]. Therefore, runtime error rates can be largely reduced by adjusting clock at cycle-level. This allows the whole system to operate at more aggressive clock frequency or supply voltage. As in previous work([16]), the compiler uses an adaptive model to predict instruction-level errors.

6 EXPERIMENTAL METHODOLOGY

Time Squeezer introduces a novel architecture and compiler co-design to exploit DTS and save energy. To empirically evaluate the proposed system, we have implemented our optimizations within an industrial strength compiler (clang-llvm [30]) and run IoT workloads on a gate-level model for an ultra-low power processor. We describe each of these components below.

6.1 Architecture Design

We develop a simulation infrastructure to study the power and performance of the proposed Time Squeezer system. The required hardware models need to capture the detailed circuit-level timing, energy, and performance of the CPU and memory system. We achieve these

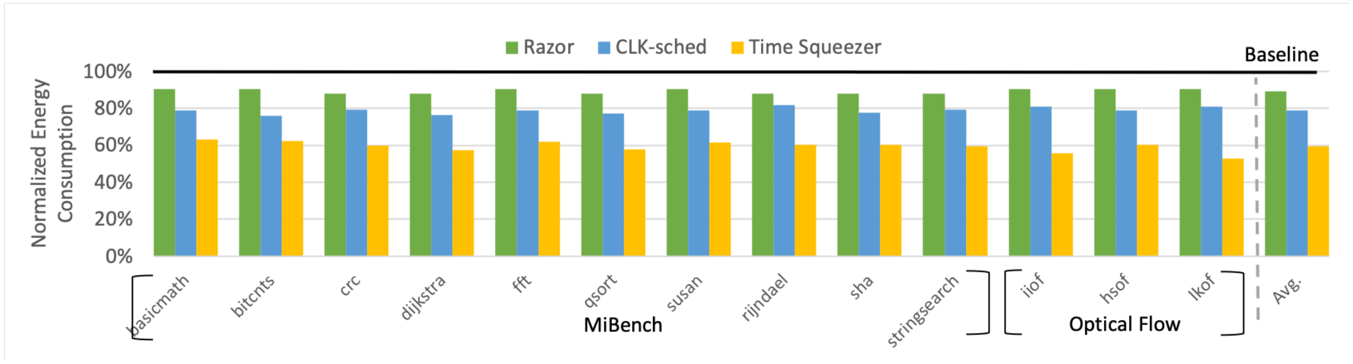


Figure 8: Comparison of energy consumption in three systems under the same performance requirement.

Technology	Low Power 55nm
Core	ARMv7 ISA, 32-bit single-issue, 6-stage Pipeline
Processor area	0.807mm^2
Core power	106.8mW @ 625MHz
Baseline Frequency	625MHz
Vdd	1.2V
PLL area	0.02 mm^2
PLL freq.	60 ∼ 1600MHz
PLL power	6.4mW @ 800MHz

Table 2: Chip Implementation Details

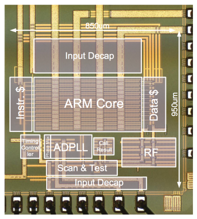


Figure 7: Die Photo.

goals by augmenting a detailed gate-level design with a number of tools and simulators.

Processor The processor pipeline models a 6-stage single issue in-order ARM pipeline with 8KB 4-way instruction and data caches. We target an ultra-low power implementation and use industry standard tools (e.g. Synopsys Design Compiler) to construct the gate-level implementation, which was also validated by our fabricated chips. This design was fabricated in a 55nm technology test chip with an implementation of the dynamic clock adjustment as shown in Figure 7 and described in Table 2. While we did not directly use the test chip in our experiments, it served as a useful frame for validation.

The design is well-optimized, and we do not expect different outcomes for alternative designs making similar energy-performance tradeoffs. On the other hand, designs that have different energy-performance goals would make a different set of design decisions at the microarchitectural level (e.g. wide issue out-of-order), macro circuit-level (e.g. high performance but power hungry adders), and gate-level (e.g. aggressive use of low vt). These types of designs offer a different set of challenges and opportunities. We are actively investigating this as a current research direction.

Simulator Our timing model for the processor uses post layout gate-level models run in Synopsys vcs [34]. Due to the impact of the proposed code transformations, extra overhead may be added to the memory system at run-time. We evaluated the overhead by closely modeling cache and memory using DRAMsim2 [40]. Detailed analysis of cache miss rate and memory footprint is described in Section 7.2. We relied on DRAMsim2 [40] to simulate the memory system (16 MB) and use the Gem5 architecture simulator [1] to fast forward to simulation regions, load checkpoint state, and validate correctness.

Validation We performed extensive critical path analysis and found that dynamic critical paths in our simulator matched those measured in the fabricated chip for all the benchmarks that we examined. We observed some slight differences which we attribute to PVT variation across chips and environmental conditions. The relative differences in DTS are large relative to the observed PVT, so we do not think that this poses a problem for Time Squeezer. More aggressive techniques that include PVT in the compiler’s delay models would likely improve the reach of this approach but are outside the scope of this paper.

We evaluate four different systems: (1) Baseline non-timing speculative six-stage single issue ARMv7 pipeline running at the nominal frequency (all results are normalized to this baseline), (2) **Razor** [12] based implementation of the baseline design, (3) a system that adds the ability to dynamically adjust the clock cycle (**CLK-sched**)(relies on compiler-generated instructions as in prior work [16]), and (4) the **Time Squeezer** co-design proposed in this paper.

6.2 Compiler and Benchmarks

We have extended the industry-strength compiler `clang` (LLVM 5.0.1 [30]) to build our Time Squeezer compiler. In particular, we have extended the middle-end of `clang` by adding a transformation pass, and we have modified the ARM back-end to implement our proposed techniques. The programmable clock generator is a multi-phase ADPLL which can scale the clock at cycle/instruction level as discussed previously. The compiler selects the conservative clock period for consecutive instructions. Note that the compiler has knowledge of the instruction schedule because the pipeline is in-order. We cross-compile and run several benchmarks from the MiBench [23] suite as well as modern Optical Flow benchmarks [45] always using the highest optimization level (i.e., -O3). Benchmarks are sampled using SimPoints [37] with interval size of 100M instructions.

7 EXPERIMENTAL EVALUATION

This section describes the empirical evaluations we performed to test Time Squeezer.

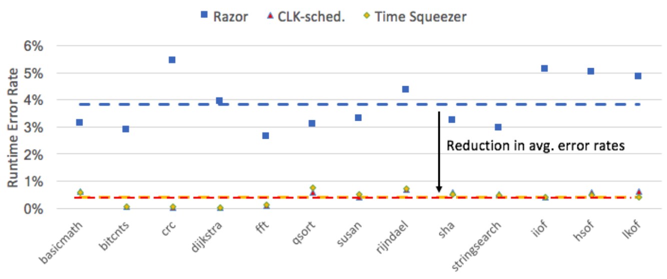


Figure 9: Only Time Squeezer is able to efficiently operate beyond the point of first failure. Note: Timing errors in all systems are detected by hardware and do not affect architecturally correct execution.

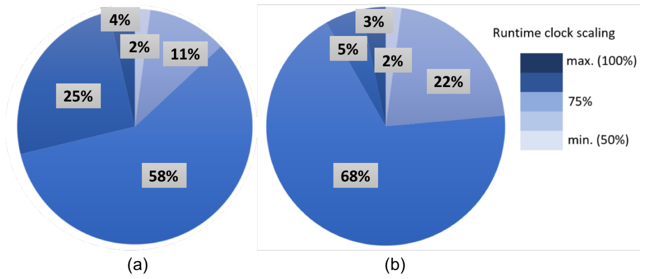


Figure 10: (a) Breakdown of clock scaling cycles for CLK-Sched and (b) Time Squeezer. Code optimizations reduce dynamic critical paths during execution, therefore a larger portion of instructions are executed under faster clocks.

7.1 Energy Evaluation

Time Squeezer aims to automatically save energy without losing performance or compromising architecturally correct execution. Under these conditions, Time Squeezer saves (on average) 40.5% of energy compared to the baseline architecture (shown in Figure 8). The CLK-sched solution, instead, saves only 21.1% of the baseline energy. Time Squeezer saves more energy than CLK-sched thanks both to the DTS-aware code generation coupled with the memory alignment described in Section 4 and to the co-design between the architecture and the compiler that enables a more energy-effective address computation. Finally, the Razor design saves only 10.9%. Razor removes a small portion of timing slack as a byproduct of eliminating safety margins. Razor's savings are blocked by its high recovery cost: as Razor optimizes more aggressively, the error rate increases and the overall recovery cost quickly becomes prohibitive. This leads Razor to operate close to the point of first failure. This is shown in Figure 9: only Time Squeezer is able to efficiently operate well beyond the point of first failure.

It is clear that Time Squeezer saves a significant amount of energy, but we conduct a deeper analysis to understand what techniques contribute most to the savings. Figure 11 shows the breakdown of energy savings for each aspect of Time Squeezer that targets DTS: the stack transformation that relies on SP rather than BP, the stack and the heap alignment, the CMP swapping transformation, and the clock adjusting at run time. We also include evaluation of a branch target precomputation optimization proposed by Hoang [24] for completeness – this is not a contribution of this paper. Figure 11 shows that all five of our proposed DTS-optimizations are necessary and contribute.

Finally, as Figure 10 shows, with Time Squeezer the system can operate at a more aggressive clock frequency for a fixed voltage. In our implementation, the common critical paths in adders are around 85% of the worst-case clock period. After code transformation, the critical paths observed at run-time are about 65%.

7.2 Overhead Analysis

Time Squeezer saves significant energy compared to the baseline and prior work despite the potential overheads that it imposes. Our analysis shows that all of the overheads are relatively small and that

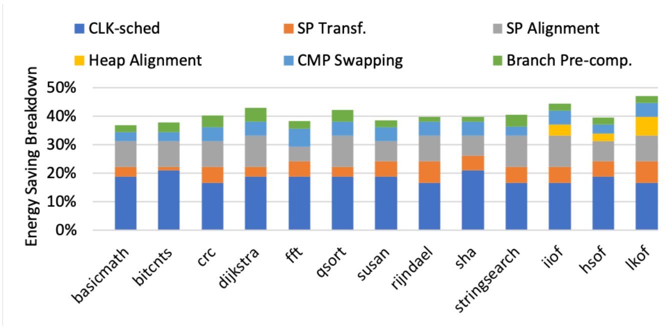


Figure 11: Breakdown of energy savings of Time Squeezing.

Time Squeezer exposes enough additional DTS that the system can exploit significant energy savings without losing performance.

The main overheads specific to Time Squeezer are: (1) an increase in dynamic instruction count, and (2) a larger memory footprint. The extra instructions executed by Time Squeezer are required for compiler directives to control the ADPLL (clock squeezing and stretching) and to force the heap alignment. On average, we find that the instruction overhead is approximately 4% due to ADPLL control and 7% due to memory alignment. The extra memory footprint is due to the stack/heap alignment which leads to some fragmentation in the program's memory. Memory footprint shows an overhead of 6.14% on average, as shown in Table 3.

Time Squeezer data alignment could lead to less compact placement of data and hence higher miss rates in caches and extra pressure in the memory system. However, the IoT workloads typical for this class of processors are typically small and frequently reuse data so that cache performance remains relatively good. In our evaluations, we simulate a core with 8KB instruction and data caches in the system, and the cache miss rates increase less than 0.35% on average.

Our results show that these overheads have negligible impact on the system performance. Moreover, our approach frees up enough additional DTS that we completely hide the overhead! Enough time can be squeezed to negate the extra instructions and slight increase in cache misses. This allows us to achieve 40.5% energy savings at the same performance target. As Table 3 shows, extra DTS analysis does add some additional time to the compilation process (less than 9% on average), but this is comparable to many other common compiler optimization such as loop hoisting performed by `clang-llvm` [30].

Benchmark	Cache Miss Rate Increase (%)	Memory Overhead (%)	Compilation Overhead (%)
basicmath	0.25	7.19	3.09
bitcnts	0.16	5.11	3.14
crc	0.45	3.41	8.16
dijkstra	0.30	4.40	9.80
fft	0.41	11.9	9.59
qsort	0.35	7.16	11.86
susan	0.30	6.85	11.39
rijndael	0.59	10.3	5.88
sha	0.41	12.6	14.06
stringsearch	0.24	4.42	5.17
iiof	0.34	6.10	11.27
hsof	0.28	7.19	6.02
lkof	0.37	11.5	9.45
Avg.	0.35	6.14	8.38

Table 3: Memory overhead.

7.3 Timing Slack Analysis

Time Squeezer saves energy by pursuing two goals: creating more DTS by changing the code and the memory layout (i.e., critical path reduction) and by exploiting newly generated DTS to shrink the clock cycles (i.e., DTS reduction). This section distinguishes the unique contributions of these two optimization goals obtained by Time Squeezer. To this end, Figure 12 shows the time spent by Time Squeezer waiting for the data (i.e., being in the circuit-level critical path) and waiting for the current clock cycle to end (i.e., remaining DTS). Figure 12 is computed by averaging the time spent waiting for these two events across all nominal (fixed) clock cycles that compose the total execution time of a given system. We performed this analysis for the four systems we have evaluated.

Time Squeezer drastically reduces the critical path compared to the baseline and prior work. The fast ADPLL design of Time Squeezer enables optimization for sub-cycle level timing slack. As Figure 12 shows, the clock cycle reductions are from two parts: timing slack reduction and critical path reduction. Compared to Razor, more timing slack (11%) is removed by applying clock scheduling. Moreover, the code transformations by the Time Squeezer compiler further shorten the critical paths by avoiding most of the carry chain stress within ALU computations and effective address generation. These instructions compose almost half the total number of instructions executed at run time (see Figure 13). We could not find any obvious optimization to target the remaining critical paths. The leftover paths are fragments related to a variety of different operations (e.g., branches, divisions, multiplications). We believe that squeezing the remaining critical paths is going to require significantly more work on top of what is described in this paper.

7.4 Sensitivity of Timing Model to PVT

Both Time Squeezer and CLK-SCHED depend on instruction-level delay models to make decisions. We performed PVT variation analysis to understand how model accuracy impacts the quality of the code.

The accuracy of the instruction-level delay model is higher in Time Squeezer. Time Squeezer relies on the instruction-level clock scheduling to reduce DTS: the compiler inserts control directives into the program so that the clock phases can be selected correspondingly at run time. The delay model reflects the correlation between instruction sequences and circuit-level critical paths. As the DTS-aware code generation of our system avoids the cases that are more likely to exercise critical paths, the instruction sub-cycle latencies become more homogeneous and, therefore, the accuracy of delay models is improved. As Figure 14 shows, the model accuracy increases from 93% to 95.6% under fixed Power-Voltage-Temperature (PVT) conditions. Moreover, even with PVT variation the accuracy slightly increases.

8 RELATED WORK

There has been a growing interest in vertical solutions for a number of important technology scaling problems [13, 41]. Many of these approaches expect circuits to be utilized outside of traditional/pessimistic operating conditions. The system still guarantees reliable computation by either correcting errors or ensuring that each instruction has enough time to compute its results. We classify prior work into two broad classes of approaches which expose some knowledge of circuit timing properties to higher layers of the system stack to improve performance or energy efficiency. The first category includes systems that exploit DTS through purely architectural support. The second category leverages compilation analysis and tailors code generation to improve operation.

8.1 DTS-aware microachitecture

Timing resilient and timing speculative microarchitectures were originally designed to maximize the potential of real silicon by eliminating the overheads associated with process, voltage, and temperature margins [8, 12, 15, 19, 42]. Although these systems have been designed to recover the aforementioned overheads, they are capable of exploiting dynamic timing slack as well. The on-line error detection and recovery mechanisms allow the voltage/frequency to be scaled beyond the margin to speculatively take advantage of any circuit-level timing slack. Moreover, the amount of timing slack can be influenced by circuit-level optimizations and the entire design can be designed from the ground up to make tradeoffs in the way that slack and timing errors appear [21, 27, 28]. The largest obstacle to achieving benefit are the recovery costs [15, 22]. Previous work has therefore applied locality properties to predict timing errors and appropriately delay the clock to allow calculations to complete [5, 49].

Under the right circumstances, it is also possible to exploit DTS without use of speculation [7, 25, 26]. To guarantee functionality, the design must only operate in an over-scaled voltage or frequency mode when it is impossible to exercise the timing critical paths which would lead to incorrect results. In some cases, the timing sensitive paths are closely coupled to specific opcodes. If workload analysis can determine that these opcodes are not present in an application, the design can operate in an over-scaled mode [7]. This approach can be low overhead, but only allows for extremely coarse grained intervals where DTS can be exploited. Fine-grained clock management can improve the reach of exploitable DTS by expanding

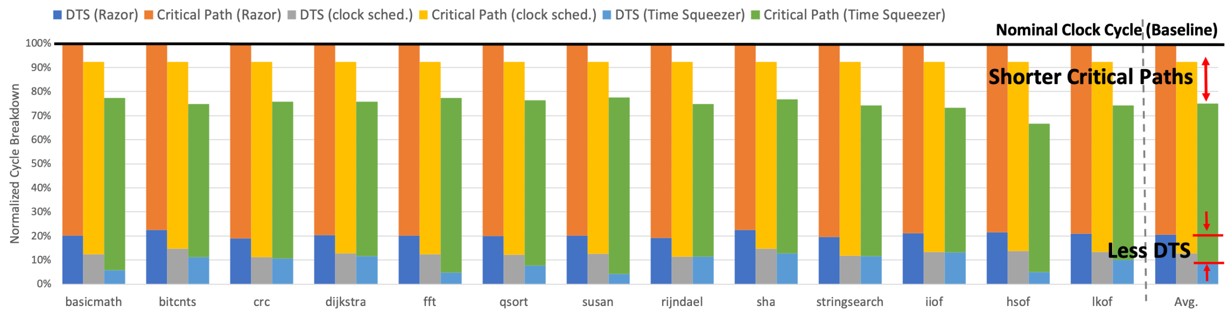


Figure 12: Breakdown of clock cycle in Razor, clock-sched., and Time Squeezer sytems. Compared to the other two, Time Squeezing both reduces the slack in clock cyles and reduce the length of critical paths during execution.

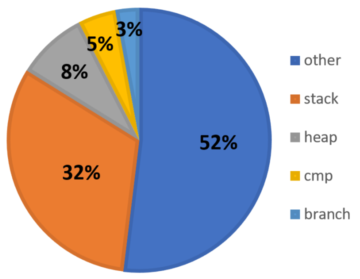


Figure 13: Breakdown of operations in program.

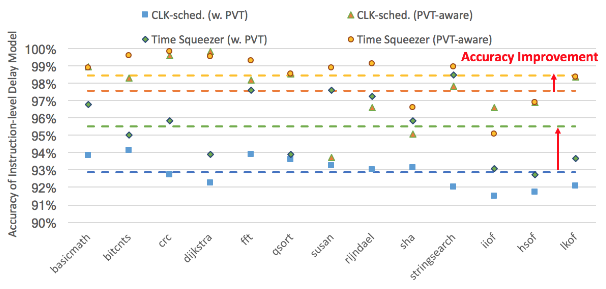


Figure 14: Due to the removal of some dynamic critical paths, Time Squeezing also improves the accuracy of instruction-level delay models.

coverage to smaller intervals where slack is found to be present. Instruction-level identification of the slack can be performed by the hardware dynamically or statically at compile time [10, 11]. Once the program regions with slack have been found, fine-grained, programmable PLL can shrink the clock when slack is present and stretch the clock when compute latencies are too long. Approaches that dynamically modify clocking behavior require more complex hardware but improve the amount of slack that can be leveraged.

8.2 DTS-aware compilers

Given the impact that instructions have on dynamic circuit-level timing, it is natural to involve the compiler in solutions that exploit timing slack. Moreover, the co-design of the compiler and architecture would seem to offer the greatest benefits. Previous work has examined the potential for considering impact of circuit-level timing at several levels of the system stack including the compiler and architecture [24]. That effort identified several simple code optimizations and associated hardware modifications that increase over-scaling opportunities for timing speculative architectures. The optimizations included NOP padding to lend extra time to critical operations, a broken increment instruction which allows modified addition operations with a truncated carry chain, branch target precomputation, and code substitution which replaces carry chain stressing subtraction operations with add instructions. Subsequent work in compiler support for timing speculation examined a wide variety of existing compiler optimizations and demonstrated that it is possible to achieve substantial benefit on timing speculative architectures when these techniques are applied [43]. Specifically, the authors examined loop unrolling, loop splitting, loop fusion, and ILP balancing as well as tuning the gcc optimization level (O0-O3). This approach does not require hardware support and leverages optimizations already present in the compiler. More recently, the compiler has been shown to be useful in identifying timing critical patterns and providing guidance to adaptive clock generators via annotations within the binary [16]. Research has also examined compiler support for reducing the impact of inductive voltage noise [38].

Our work differs from prior efforts in several ways. First we take a broader view of reasons why timing slack appears and this allows us to recognize the impact of data as well as instructions. Specifically, our techniques are built around recognizing how specific types of data will impact the instruction-level timing. Second, we introduce use of data management as a powerful tool in shaping the way that the program needs to compute effective addresses. Prior work at the compiler level has examined the impact of the carry chain on some types of computation [24], but did not consider address computations and placement of data. This represents a significantly higher degree of complexity and is a unique contribution of this paper.

Finally, we take a holistic view of the system which allows us to create additional DTS opportunities in both the hardware and code generation that would not otherwise be possible.

9 LIMITATIONS AND FUTURE WORK

The above section shows the results of the Time Squeezer system. In this section, the discussion includes some limitations and future works.

Generality and limitations One of the key ideas in the Time Squeezer system is that the compiler needs to be aware of the critical paths in the architecture design. In this work, we focus on the most common operations in IoT applications/benchmarks, including effective address generation, value comparison, addition and subtraction, which comprise a majority of the critical execution. The next class of instructions to address would be less frequent primitive operations (e.g. multiplication, division) – targets for future work. Devices in the IoT space tend to use in-order shallow pipelines for energy efficiency and fit the proposed technique very well, for example the one shown in [7]. In these types of processors, execution logic is responsible for many of the critical paths and the Time Squeezer techniques introduced here can be impactful. However, as the hardware complexity increases, critical paths will become prominent in many other parts of the pipeline, especially in control logic. In that case, Time Squeezer would need to consider a wider range of instruction and data interactions to obtain similar benefits.

Pathological behavior A worst case would likely arise if there were a large number of objects that forced poor alignment. We believe that this will be unlikely for the systems and workloads targeted in this work. In particular, it is unlikely that programs written in commercially ubiquitous object oriented (OO) languages will lead to higher overhead. Applications written in some of the most common OO languages, namely Java or C#, characteristically instantiate mostly small objects. Hence, the overhead of aligning small objects to a power of two will be small. Moreover, our compiler can disable this alignment for the few large objects that will be allocated avoiding these pathological cases. This is similar to the ability to disable the power-of-two alignment for large data structures in C programs. While our compiler is capable of disabling this alignment per allocation, we did not see the need for this for the benchmarks we targeted in this paper. Table 3 shows that the memory overhead is small even if we have aligned all memory objects and stack variables.

Libraries, third-party code, and interoperability In the IoT space, it is common for systems to be hardware/software co-designed and in some cases run on bare metal (without benefit of an operating system). We correspondingly assume that all the code is generated by our compiler and that the designer has complete control of the entire system stack. On the other hand, in the case where third party software (e.g. external libraries) is important, the cleanest solution would be to extend the ISA to support two kinds of effective address computation (1) [base+displacement] and (2) [base|displacement] and assume that our approach is used to manage heap data. Note that third party software will still be able to reference objects in memory (e.g. heap, locals, and spills) using option (1), but that this may trigger (recoverable) timing errors affecting performance but not correctness. This is a simple extension to our work, which does not require any fundamental change.

Future work Our work focused on processors for tiny embedded devices because they simultaneously have extremely tight power constraints and no obvious excess resources which can be sacrificed without losing performance. In these devices the system designer typically has control over the hardware/software stack and hence there is considerable room for cross-layer optimizations like the one we propose.

In future work, we plan to explore dynamism in the system in two important ways. First, we will investigate more aggressive out-of-order cores. In a deep, well-balanced pipeline, there are critical paths in every stage. We expect that timing critical execution paths addressed in this paper will remain important, but they will be accompanied by other paths related to extracting ILP and supporting speculation. We anticipate that our proposed techniques will remain valid, but the compiler will likely need to consider a wider variety of code optimizations to reach its full potential. Second, we will examine the role that just in time (JIT) compilation can play in managing dynamic timing slack. JIT compilation should offer a number of interesting opportunities to respond to changing patterns in the workload and environment.

10 CONCLUSION

As we near the end of Moore's Law scaling, there has been growing interest in clawing back energy savings through any means necessary. Dynamic timing slack presents interesting opportunities for realizing some of these savings. In this paper, we introduced compiler and architecture co-design where the compiler squeezes time at the sub-cycle level to create new amounts of dynamic timing slack. Our evaluations show that the architecture can efficiently exploit the slack to produce 40.5% energy savings on average. This results holds particular promise for the emerging class of tiny embedded devices.

ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their helpful feedback. We thank Tianyu Jia and Jie Gu for their contributions to the silicon fabrication and measurement. We thank Enrico A. Deiana for creating the TimeSqueezer logo used in Figure 3. This work was partially supported by the National Science Foundation of the United States under grant # CCF-1618065.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[2] Shekhar Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Ieee Micro* 25, 6 (2005), 10–16.

[3] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. 2003. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th annual Design Automation Conference*. ACM, 338–342.

[4] Peter D Bradley. 2006. An ultra low power, high performance medical implant communication system (MICS) transceiver for implantable devices. In *Biomedical Circuits and Systems Conference, 2006. BioCAS 2006. IEEE*. IEEE, 158–161.

[5] Koushik Chakraborty, Brennan Cozzens, Sanghamitra Roy, and Dean M Ancajas. 2013. Efficiently tolerating timing violations in pipelined microprocessors. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 102.

[6] Sriram Cherukuri, Krishna K Venkatasubramanian, and Sandeep KS Gupta. 2003. Biosec: A biometric based approach for securing communication in wireless networks of biosensors implanted in the human body. In *Parallel Processing Workshops, 2003. Proceedings. 2003 International Conference on*. IEEE, 432–439.

[7] Hari Cherupalli, Rakesh Kumar, and John Sartori. 2016. Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 671–681.

[8] Mihir Choudhury, Vikas Chandra, Kartik Mohanram, and Robert Aitken. 2010. TIMBER: Time borrowing and error relaying for online timing error resilience. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 1554–1559.

[9] Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay, and Andreas Burg. 2015. Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment. *Proceedings of the 2015 Design, Automation & Test in Europe* (2015), 381–386.

[10] Jeremy Constantin, Lai Wang, Georgios Karakonstantis, Anupam Chattopadhyay, and Andreas Burg. 2015. 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*. EDA Consortium, 381–386.

[11] Jeremy Hugues-Felix Constantin, Andrea Bonetti, Adam Shmuel Teman, Thomas Christoph Müller, Lorenz Flavio Schmid, and Andreas Peter Burg. 2016. DynOR: A 32-bit microprocessor in 28 nm FD-SOI with cycle-by-cycle dynamic clock adjustment. In *Esscirc Conference 2016*. Ieee, 261–264.

[12] S. Das, C. Tokunaga, S. Pant, W. H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw. 2009. RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. *IEEE Journal of Solid-State Circuits* 44, 1 (Jan 2009), 32–48. https://doi.org/10.1109/JSSC.2008.2007145

[13] Marc De Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 497–508.

[14] Touby Drew and Maria Gini. 2006. Implantable medical devices as agents and part of multiagent systems. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM, 1534–1541.

[15] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, Trevor Mudge, Beal Ave, and Ann Arbor. 2003. Razor : A Low-Power Pipeline Based on Circuit-Level Timing Speculation. December (2003).

[16] Yuanbo Fan, Tianyu Jia, Jie Gu, Simone Campanoni, and Russ Joseph. 2018. Compiler-guided Instruction-level Clock Scheduling for Timing Speculative Processors. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, New York, NY, USA, Article 40, 6 pages. https://doi.org/10.1145/3195970.3196013

[17] Maria Fazio, Maurizio Paone, Antonio Puliafito, and Massimo Villari. 2012. Heterogeneous sensors become homogeneous things in smart cities. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*. IEEE, 775–780.

[18] Dario Floreano and Robert J Wood. 2015. Science, technology and the future of small autonomous drones. *Nature* 521, 7553 (2015), 460.

[19] Matthew Fojtik, David Fick, Yejoong Kim, Nathaniel Pinckney, David Money Harris, David Blaauw, and Dennis Sylvester. 2013. Bubble razor: Eliminating timing margins in an ARM cortex-M3 processor in 45 nm CMOS using architecturally independent error detection and correction. *IEEE Journal of Solid-State Circuits* 48, 1 (2013), 66–81.

[20] Ali Murat Gok and Nikos Hardavellas. 2017. VaLHALLA: Variable Latency History Aware Local-carry Lazy Adder. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, 17–22.

[21] Brian Greskamp, Lu Wan, Ulya R Karpuzcu, Jeffrey J Cook, Josep Torrellas, Deming Chen, and Craig Zilles. 2009. Blueshift: Designing processors for timing speculation from the ground up.. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*. IEEE, 213–224.

[22] Meeta S Gupta, Jude A Rivers, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2009. Tribeca: design for PVT variations with local recovery and fine-grained adaptation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 435–446.

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01)*. IEEE Computer Society, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

[24] Giang Hoang, Robby Bruce Findler, and Russ Joseph. 2011. Exploring circuit timing-aware language and compilation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 345–356.

[25] Tianyu Jia, Yuanbo Fan, Russ Joseph, and Jie Gu. 2016. Exploration of associative power management with instruction governed operation for ultra-low power design. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[26] T. Jia, R. Joseph, and Jie Gu. 2017. Greybox design methodology: A program driven hardware co-optimization with ultra-dynamic clock management. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062255

[27] Andrew B Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. 2010. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 1–11.

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

[29] Kaivan Karimi and Gary Atkinson. 2013. What the Internet of Things (IoT) needs to become a reality. *White Paper, FreeScale and ARM* (2013), 1–16.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[31] Gai Liu, Ye Tao, Mingxing Tan, and Zhiru Zhang. 2014. CASA: correlation-aware speculative adders. In *Low Power Electronics and Design (ISLPED), 2014 IEEE/ACM International Symposium on*. IEEE, 189–194.

[32] Krishna T Malladi, Frank A Nothaft, Karthika Periyathambi, Benjamin C Lee, Christos Kozyrakis, and Mark Horowitz. 2012. Towards energy-proportional datacenter memory with mobile DRAM. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 37–48.

[33] Online. [n. d.]. *Edge TPU*. Available from https://cloud.google.com/edge-tpu.

[34] Online. [n. d.]. *Synopsys VCS User Guide*. Available from https://www.synopsys.com/verification/simulation/vcs.html.

[35] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. 2017. Harnessing voltage margins for energy efficiency in multicore CPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 503–516.

[36] Dinesh Patil, Omid Azizi, Mark Horowitz, Ron Ho, and Rajesh Ananthraman. 2007. Robust energy-efficient adder topologies. In *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on*. IEEE, 16–28.

[37] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Perform. Eval. Rev.* 31, 1 (June 2003), 318–319. https://doi.org/10.1145/885651.781076

[38] Vijay Janapa Reddi, Simone Campanoni, Meeta S Gupta, Michael D Smith, Gu-Yeon Wei, David Brooks, and Kim Hazelwood. 2010. Eliminating voltage emergencies via software-guided code transformations. *ACM Transactions on Architecture and Code Optimization (TACO)* 7, 2 (2010), 12.

[39] Jeremy Rifkin. 2014. *The zero marginal cost society: The internet of things, the collaborative commons, and the eclipse of capitalism*. St. Martin's Press.

[40] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Archit. Lett.* 10, 1 (Jan. 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[41] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 164–174.

[42] Smruti Sarangi, Brian Greskamp, Abhishek Tiwari, and Josep Torrellas. 2008. EVAL: Utilizing processors with variation-induced timing errors. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. IEEE, 423–434.

[43] John Sartori and Rakesh Kumar. 2012. Compiling for energy efficiency on timing speculative processors. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1301–1308.

[44] Robert R Schaller. 1997. Moore's law: past, present and future. *IEEE spectrum* 34, 6 (1997), 52–59.

[45] D. Sun, S. Roth, and M. J. Black. 2010. Secrets of optical flow estimation and their principles. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2432–2439. https://doi.org/10.1109/CVPR.2010.5539939

[46] Thomas N Theis and H-S Philip Wong. 2017. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering* 19, 2 (2017), 41–50.

[47] Ajay K Verma, Philip Brisk, and Paolo Ienne. 2008. Variable latency speculative addition: A new paradigm for arithmetic circuit design. In *Proceedings of the conference on Design, automation and test in Europe*. ACM, 1250–1255.

[48] Joseph Wei. 2014. How Wearables Intersect with the Cloud and the Internet of Things: Considerations for the developers of wearables. *IEEE Consumer Electronics Magazine* 3, 3 (2014), 53–56.

[49] Jing Xin and Russ Joseph. 2011. Identifying and predicting timing-critical instructions to boost timing speculation. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 128–139.

[50] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. 2014. Internet of things for smart cities. *IEEE Internet of Things journal* 1, 1 (2014), 22–32.